Date: April 2020

OBJECT MANAGEMENT GROUP®

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

Version 1.2

OMG Document Number: formal/20-04-02

Normative reference: http://www.omg.org/spec/UCM/1.1

Associated Normative Machine Consumable Files:

https://www.omg.org/spec/UCM/20190501/ucm_base.xmi
https://www.omg.org/spec/UCM/20190501/ucm_base.xsd
https://www.omr.org/spec/UCM/20190501/ucm_base.dtd
https://www.omg.org/spec/UCM/20190501/ucm.idl
https://www.omg.org/spec/UCM/20190501/ucm_core.xml
https://www.omg.org/spec/UCM/20190501/ucm_ext_exec.xml
https://www.omg.org/spec/UCM/20190501/ucm_ext_interac.xml
https://www.omg.org/spec/UCM/20190501/ucm_std_am.xml
https://www.omg.org/spec/UCM/20190501/ucm_timer.xml
https://www.omg.org/spec/UCM/20190501/ucm_lang--cpp.xml

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 https://www.omg.org, under Documents, Report a Bug/Issue (https://www.omg.org/technology/agreement.)

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 https://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:

https://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.

Courier - 10 pt. Bold: 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 https://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 are also 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.

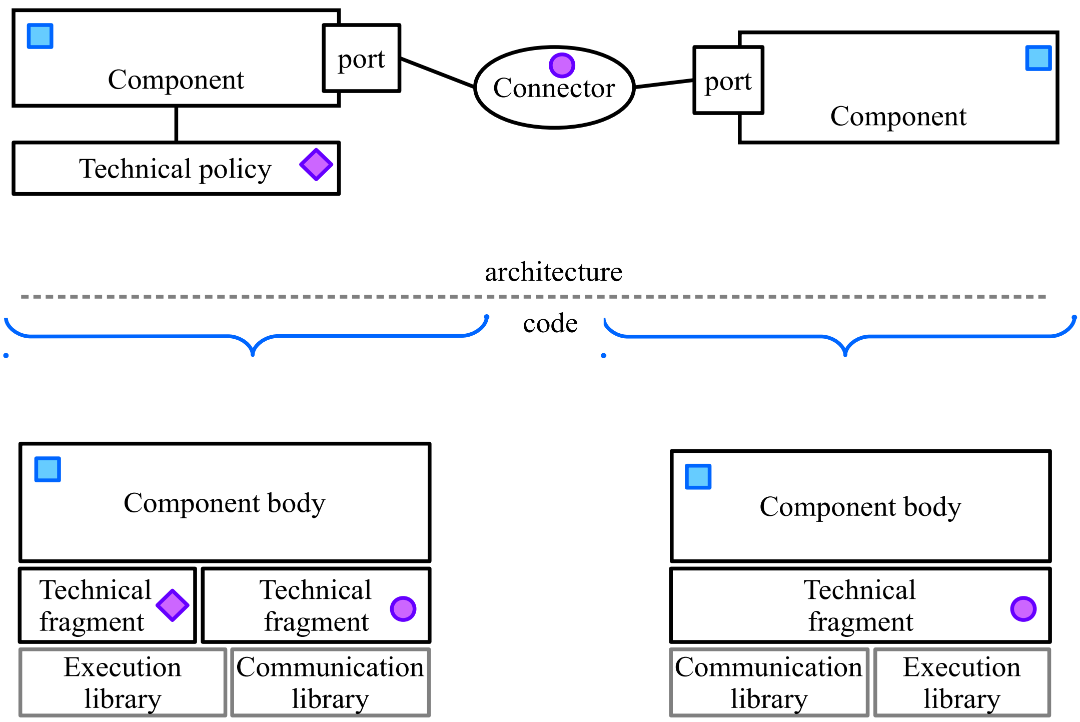

Figure 1: From components to software

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 used by the components, just like a programming language standard library is used by the 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 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 are 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 are 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.

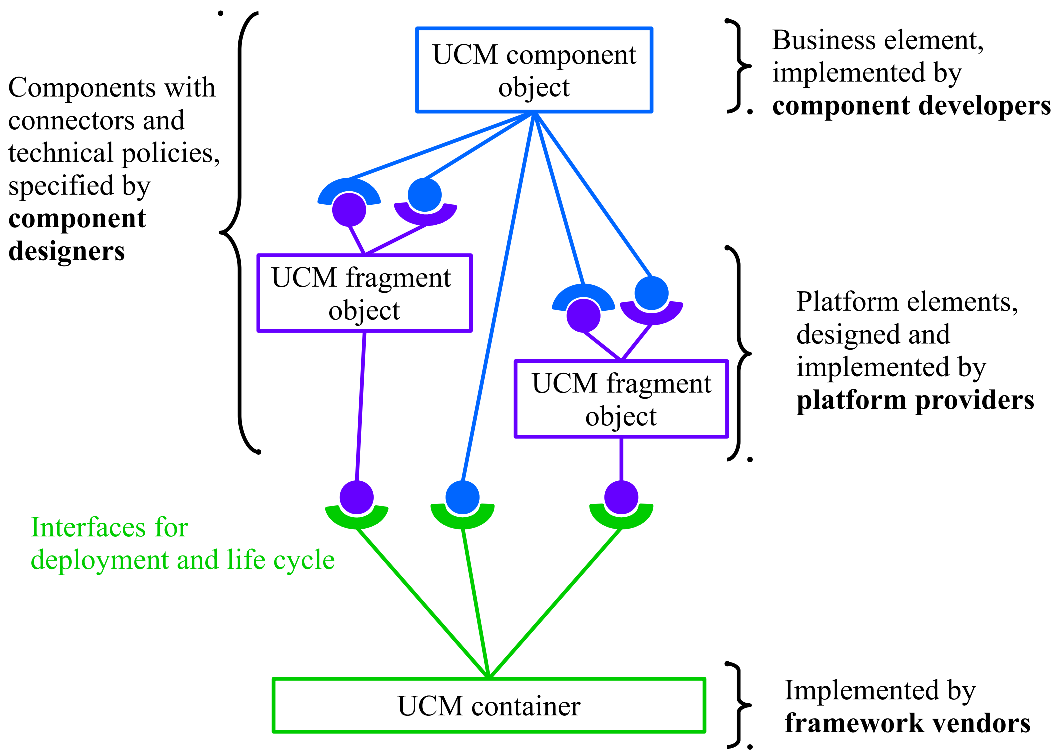

Figure 2: Component implementation architecture and its integration within a UCM infrastructure

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 (§ 13.1), which address the basic interaction and technical policy APIs. All UCM platforms shall support the UCM core specifications. Any UCM platform shall support the execution of any component that conforms with the UCM core specification (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 (§ 9.3 and 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 shall 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 shall 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 may 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 13.

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 may have attributes. Attributes are functional parameters that may 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 13.1, 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 shall support the UCM PIM defined in section 9. They also shall at least ship with implementations of the core UCM platform specifications described in section 13.1. Implementation code shall conform to the standard programming model (section 14).

UCM frameworks may 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. Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- IETF RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels", March 1997.Available from http://ietf.org/rfc/rfc2119.
- Interface Definition Language (IDL) 4.0 - formal/2016-04-02
- Extensible Markup Language (XML) 1.1, September 2006. Available from https://www.w3.org/TR/2006/REC-xml11-20060816/
- ISO/IEC 14882:2003, Information Technology - Programming languages – C++.

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 may also contain **configuration parameters** to specify nonfunctional settings (e.g. queue size, communication protocol).

Atomic component implementations may 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 may define APIs to interact with the component, and may 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

UML: 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)

This page intentionally left blank.

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 ptc/2019-04-05.

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

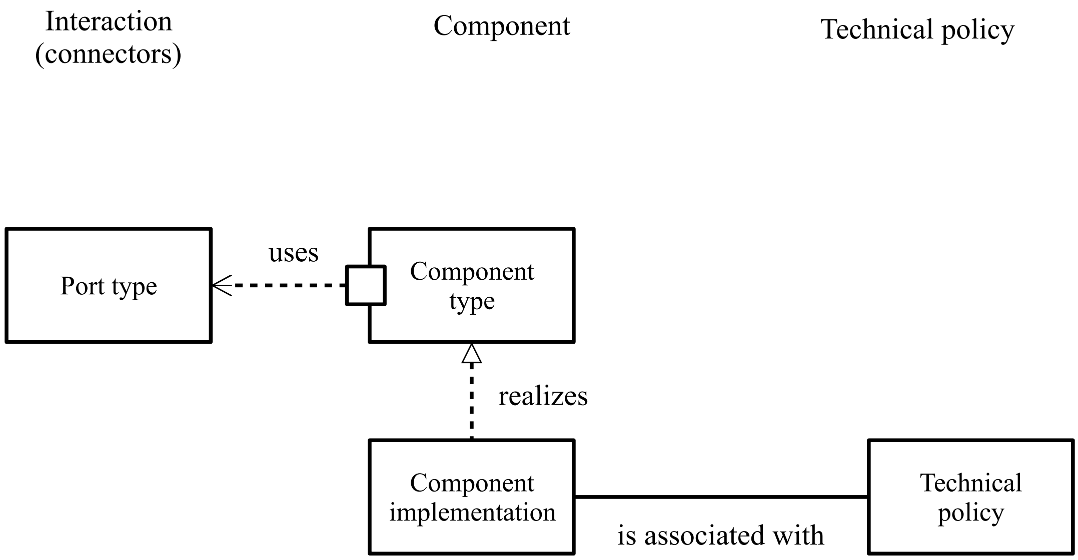

Figure 3: Relationship between components, connectors and technical policies

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 may 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 may be manipulated by the business code nested in components. Configuration parameters are nonfunctional elements: they should be processed by framework tools but should not be seen by the business code. Properties are used to decorate functional elements; though they are not manipulated by the business code, they are like 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

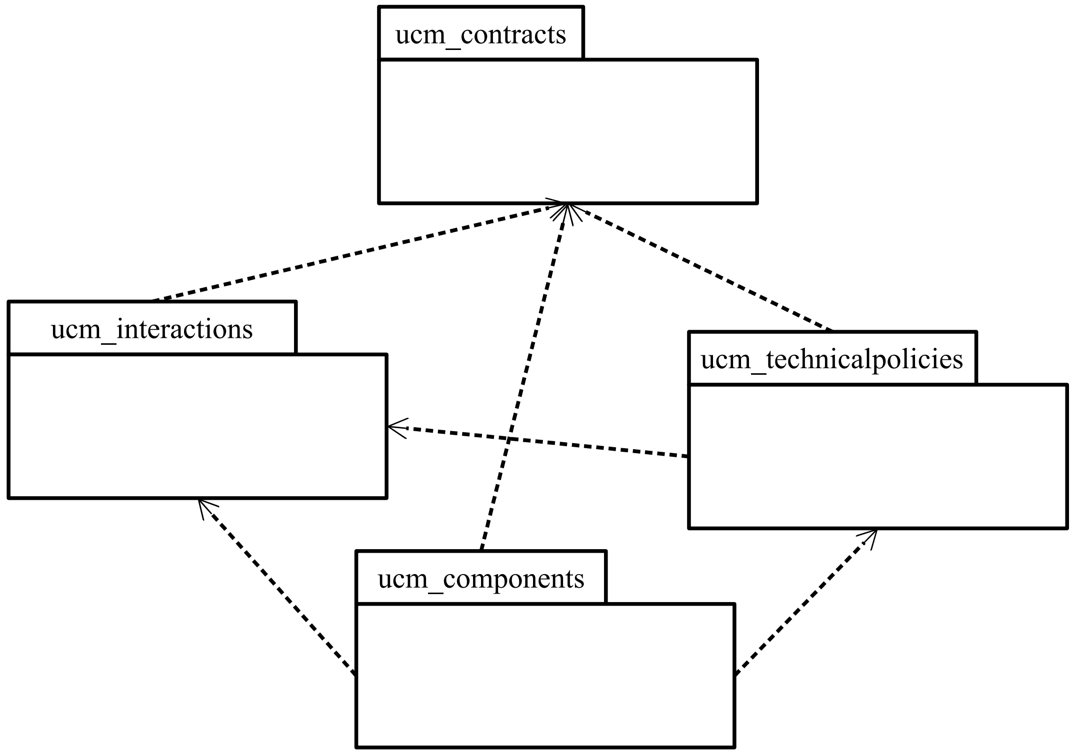

Figure 4: Main packages in the UCM meta-model

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 may 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 (§ 13.1 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

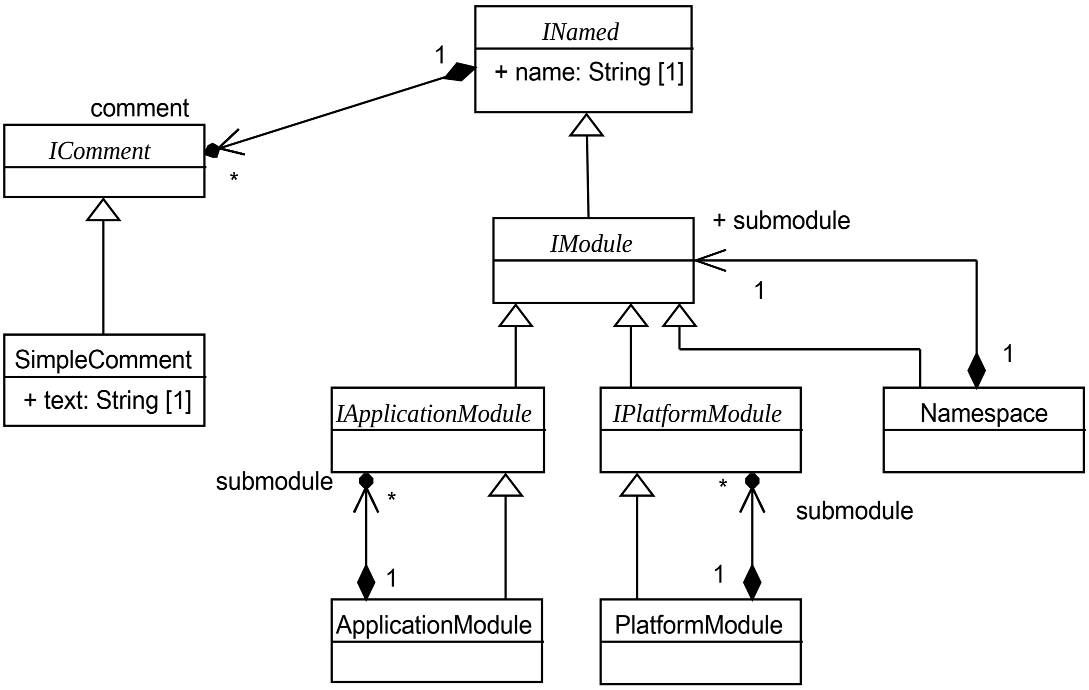

Figure 5: Base classes

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:

- name: String [1] (owned)
- comment: IComment [0…*] (owned)

All named UCM entities derive from class INamed. The absolute name of an INamed entity consists of the list of the names of its parent named entities, starting from the model root, followed by the name of the entity itself. Each named UCM entity of a given set of UCM models shall have a unique absolute name. The only exceptions are namespaces, which can be declared several times (see section 9.1.4.9).

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] (owned)

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 may 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 are used to gather several component and contract modules.

- submodule: IApplicationModule [0…*] (owned)

9.1.4.8 PlatformModule (IPlatformModule)

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

- submodule: IPlatformModule [0…*] (owned)

9.1.4.9 Namespace (IModule)

Class Namespace is used to split a given logical module hierarchy into several files. Unlike other UCM named entities, two namespaces may have the same absolute name, but only if they are declared in different files. A namespace shall not be empty: it must contain a submodule or another namespace.

- submodule: IModule [1] (owned)

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 shall 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 shall be used as hooks to support meta-model extensions.

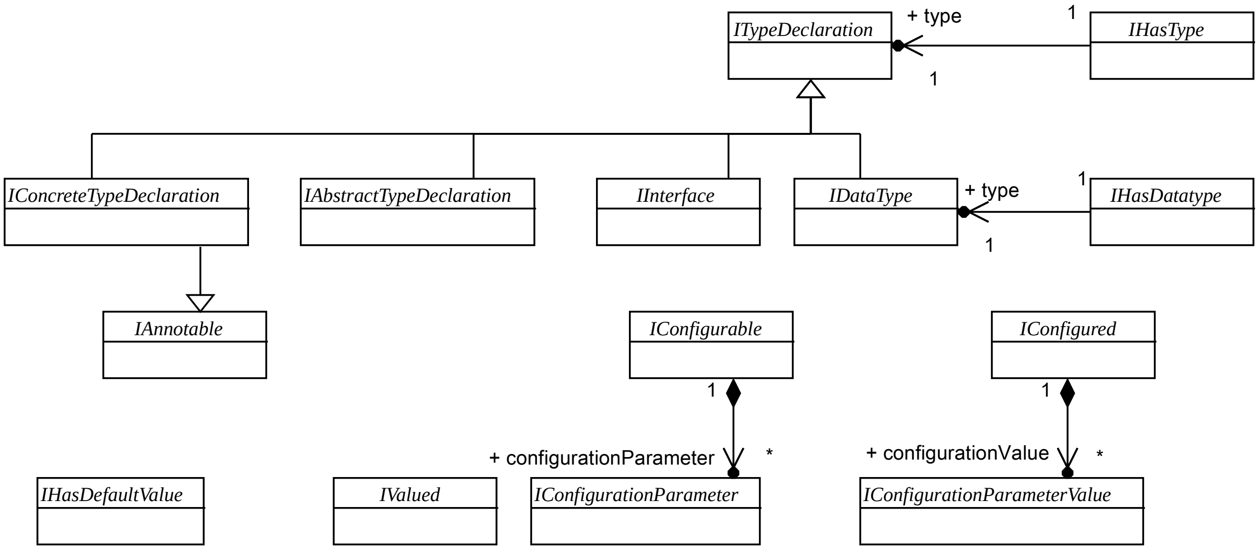

Figure 6: Abstract base classes

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.

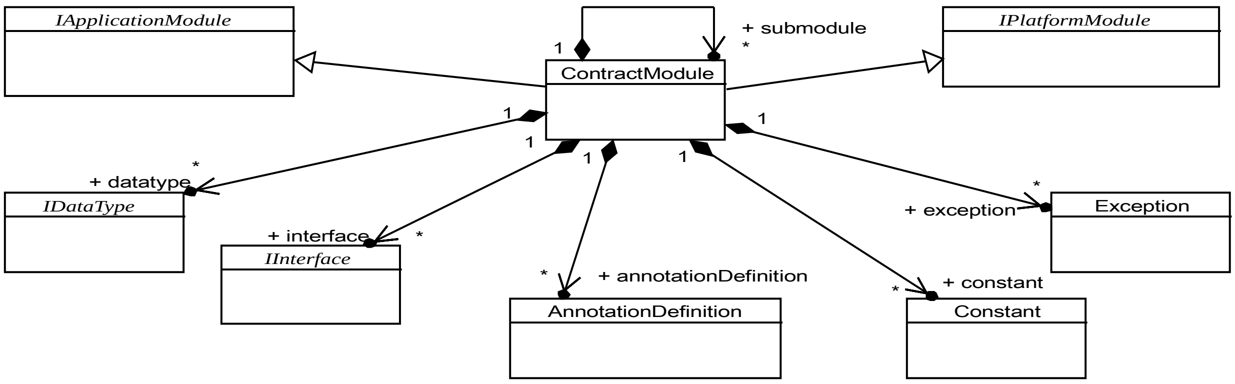

Figure 7: UCM contract base declarations

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 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 entities that are typed by 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 reference either data types or interfaces (typically, method parameters).

- type: ITypeDeclaration [1]

9.2.2.6 IValued

Abstract class IValued is the common ancestor for data type declarations that have a value.

- value: String [1] (owned)

Field value is a plain string. IDL syntax shall be used to specify values. See formal/18-01-05.

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] (owned)

9.2.2.8 IAnnotable

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

- annotation: Annotation [0…*] (owned)

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.

- lower: Integer [1] (owned)

- upper: Integer [1] (owned)

Fields lower and upper specify the minimum and maximum expected number of configuration parameter values. This means that, if the upper value is strictly greater than 1, then several IConfigurationParameterValue may be associated with the IConfigured entity. If lower is set to 0, then the configuration parameter is optional.

The value of field upper shall be greater than or equal to the value of field lower; however, if there is no upper limit, the value of field upper shall be set to -1. The value of field lower shall be greater than or equal to 0.

The default values of fields lower and upper is 1.

9.2.2.12 IConfigurable

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

- configurationParameter: IConfigurationParameter [0…*] (owned)

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 may specify configuration parameter values. See section 9.4).

- configurationValue: IConfigurationParameterValue [0…*] (owned)

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 may be nested in other contract modules to create hierarchies. Contract module also contain annotation definitions (§ 9.2.9). Fields are:

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

Contract modules are comparable to 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.

9.2.3.1 IStandardDataType (IConcreteTypeDeclaration, IDataType)

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

9.2.3.2 IPrimitiveDataType

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

9.2.3.3 PrimitiveInteger (IStandardDataType, IPrimitiveDataType, IDiscreteType, IScalarType)

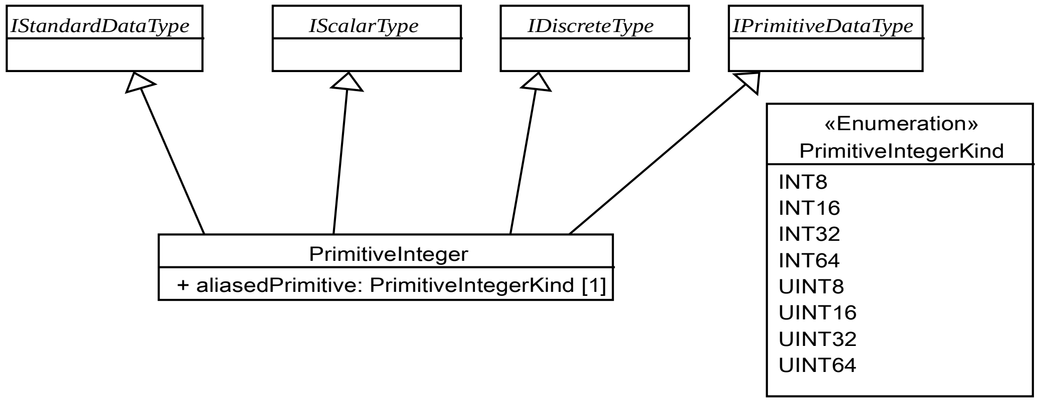

Figure 8: UCM primitive integers

Class PrimitiveInteger corresponds to all kinds of integer types.

- aliasedPrimitive: PrimitiveIntegerKind [1] (owned)

Enumeration PrimitiveIntegerKind has these values: INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64. The default value of field aliasedPrimitive is INT16.

UCM integer type ranges are detailed in the following table:

UCM integer type	Lower bound	Upper bound	IDL equivalent
INT8	-2^{7}	2^{7}-1	int8 (in IDL 4.2)
INT16	-2^{15}	2^{15}-1	short
INT32	-2^{31}	2^{31}-1	long
INT64	-2^{63}	2^{63}-1	long long
UINT8	0	2^{8} - 1	uint8 (in IDL 4.2)
UIN16	0	2^{16}-1	unsigned short
UINT32	0	2^{32}-1	unsigned long
UINT64	0	2^{64}-1	unsigned long long

9.2.3.4 PrimitiveFloat (IStandardDataType, IPrimitiveDataType, IScalarType)

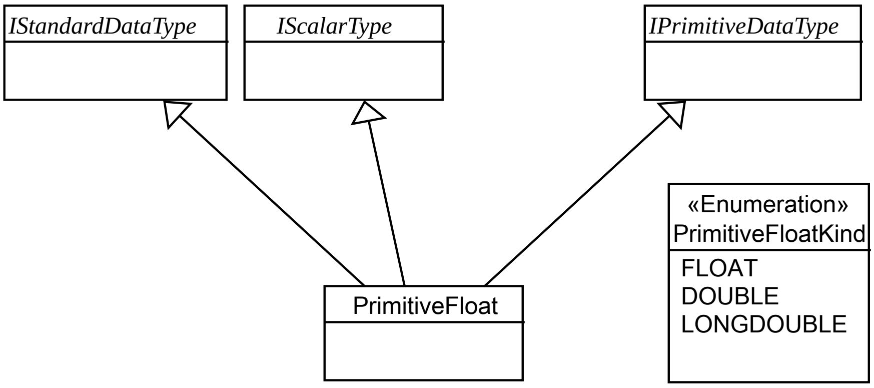

Figure 9: UCM floating point numbers

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

- aliasedPrimitive: PrimitiveFloatKind [1] (owned)

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).

9.2.3.5 PrimitiveChar (IStandardDataType, IPrimitiveDataType, IDiscreteType, IScalarType)

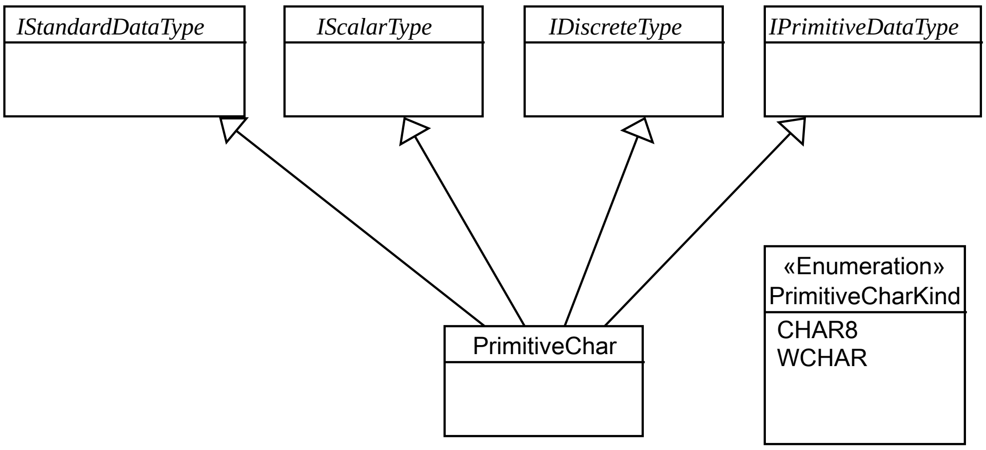

Figure 10: UCM primitive characters

Class PrimitiveChar corresponds to all kinds of character types.

- aliasedPrimitive: PrimitiveCharKind [1] (owned)

Enumeration PrimitiveCharKind has two values: CHAR8 and WCHAR.

CHAR8 corresponds to an ASCII 8 bit character or a UTF-8 character. WCHAR corresponds to a wide character, the exact implementation of which is language and compiler dependent.

9.2.3.6 PrimitiveBoolean (IStandardDataType, IDiscreteType, IPrimitiveDataType, IScalarType)

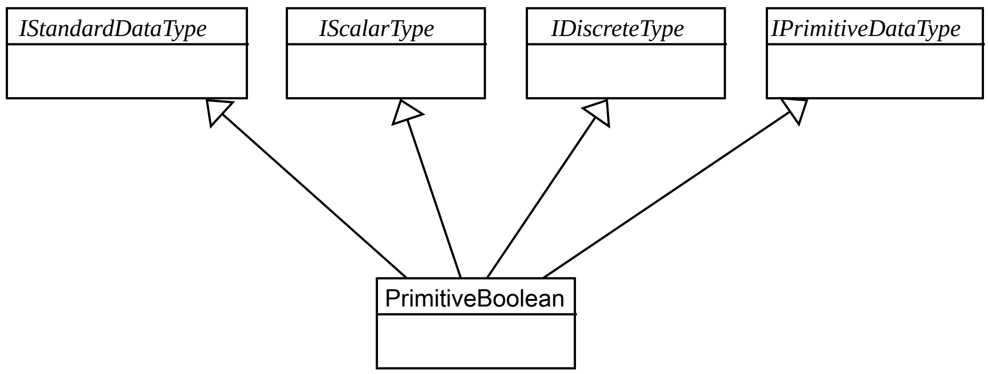

Figure 11: UCM primitive boolean

Class PrimitiveBoolean corresponds to the boolean type.

9.2.3.7 PrimitiveOctet (IStandardDataType, IPrimitiveDataType)

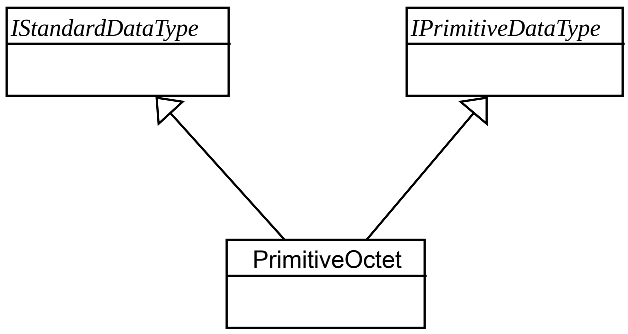

Figure 12: UCM primitive octet

Class PrimitiveOctet corresponds to an 8 bit buffer element, like the IDL octet type.

9.2.4 Standard data types: complex types

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

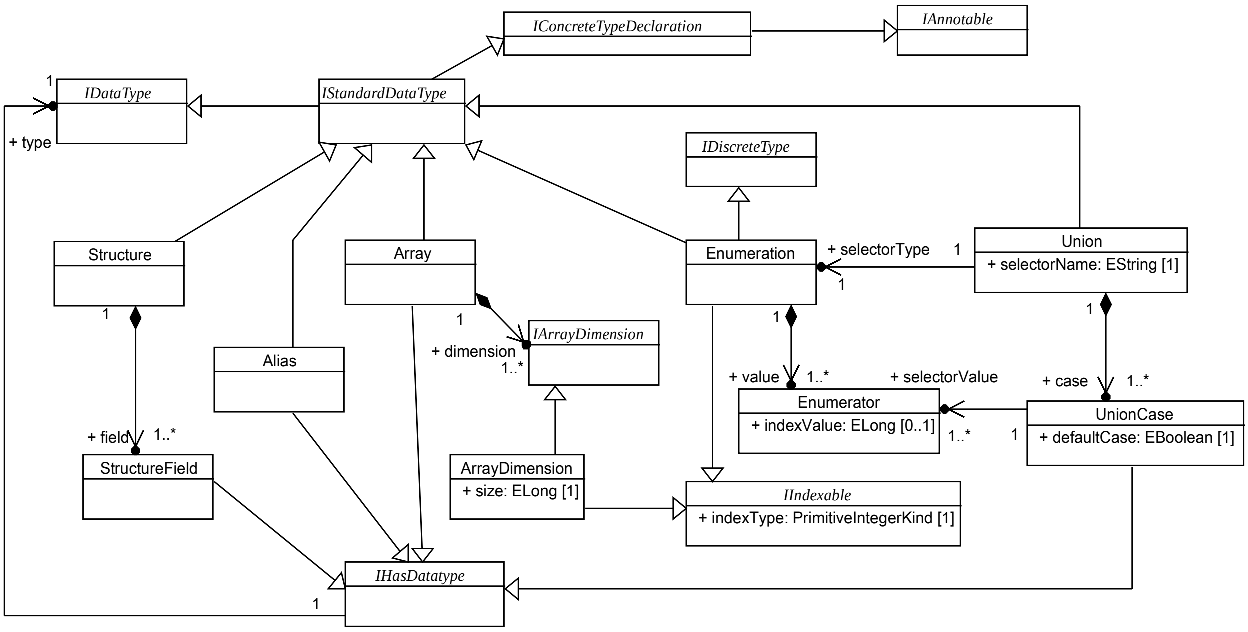

Figure 13: UCM complex data types

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] (owned)

Indexable types are indexed by an integer. The default value of field indexType is UINT32.

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 shall have an identifier and a type.

- field: StructureField [1…*] (owned)

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 takes 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] (owned)

- selectorType: Enumeration [1]

- case: UnionCase [1…*] (owned)

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] (owned)

Cases shall 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 shall 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 should 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 shall refer to an integer type from which it shall take its values. An enumeration declares at least one enumerator that describes the accepted values for the enumeration.

- value: Enumerator [1…*] (owned)

9.2.4.8 Enumerator (INamed)

An enumerator corresponds to a value literal.

- indexValue: Long [1] (owned)

The index value shall 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 may be multidimensionnal, each dimension having potentially different index types.

- dimension: IArrayDimension [1…*] (owned)

9.2.4.10 IArrayDimension

Abstract class IArrayDimension is meant to allow meta-model extensions. For example, the UCM meta-model may 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] (owned)

Size is a long integer. As ArrayDimension inherits from IIndexable, size shall 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.

9.2.5.1 IResizable

Abstract class IResizable is used for types that behave as collections of objects the size of which may 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 may 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] (owned)

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

9.2.5.2 StringType (IStandardDataType, IResizable)

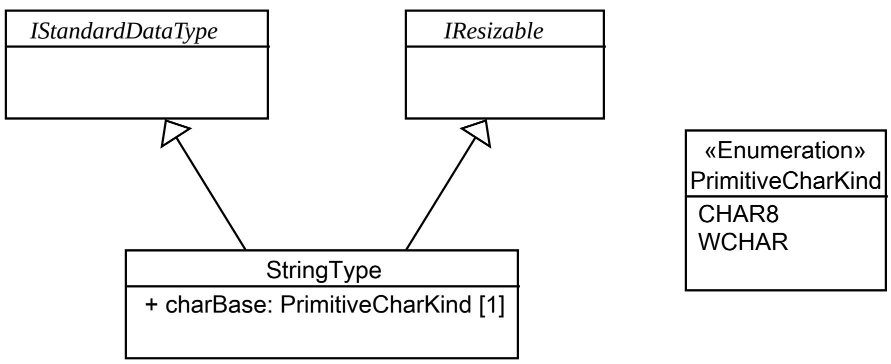

Figure 14: UCM string type

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

- charBase: PrimitiveCharKind [1] (owned)

9.2.5.3 NativeType (IStandardDataType, IResizable)

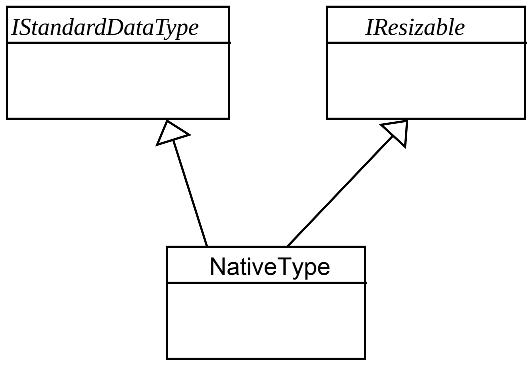

Figure 15: UCM native type

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 (IStandardDataType, IHasDataType, IResizable, IIndexable)

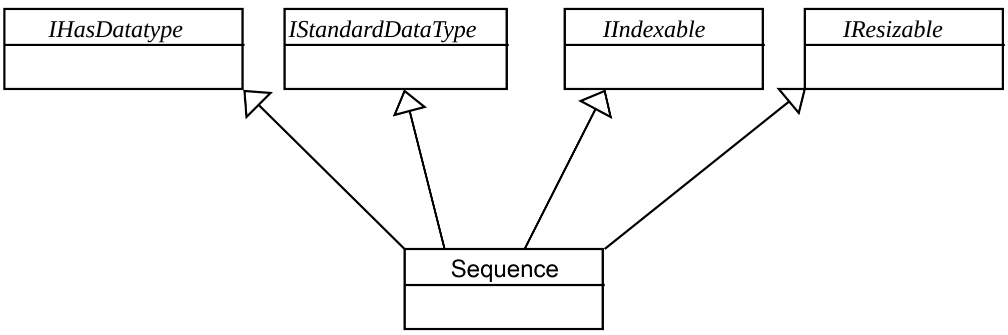

Figure 16: UCM sequence

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

Sequences are like one-dimension arrays with a variable size. Their sizes are bounded. Unbounded sequences shall have their maxSize field set to -1 or less.

9.2.6 Constants

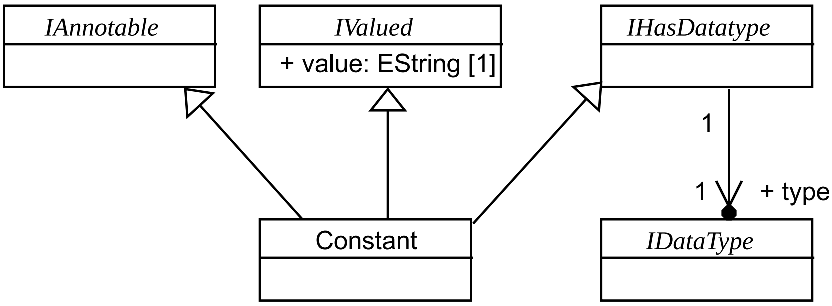

Figure 17: UCM constants

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 formal/18-01-05.

9.2.7 Interfaces, methods and exceptions

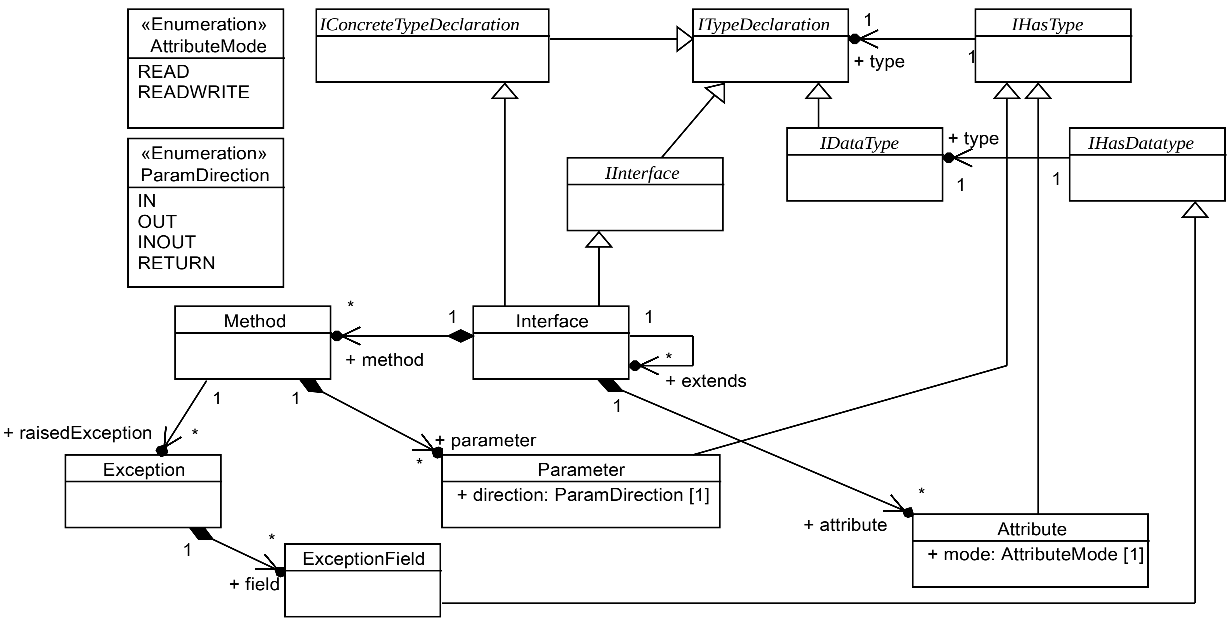

Figure 18: UCM interfaces

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 shall 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 may 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 shall 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 (IInterface, IConcreteTypeDeclaration)

- extends: Interface [0…*]
- attribute: Attribute [0…*] (owned)
- method: Method [0…*] (owned)

An interface may 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…*] (owned)
- raisedException: Exception [0…*]

9.2.7.3 Parameter (INamed, IHasType)

- direction: ParamDirection [1] (owned)

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 shall have at most one “return” parameter.

9.2.7.4 Attribute (INamed, IHasType, IAnnotable, IHasDefaultValue)

- mode: AttributeMode [1] (owned)

AttributeMode is an enumeration with the following values: read, readwrite.

9.2.7.5 Exception (INamed)

- field: ExceptionField [0…*] (owned)

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 .

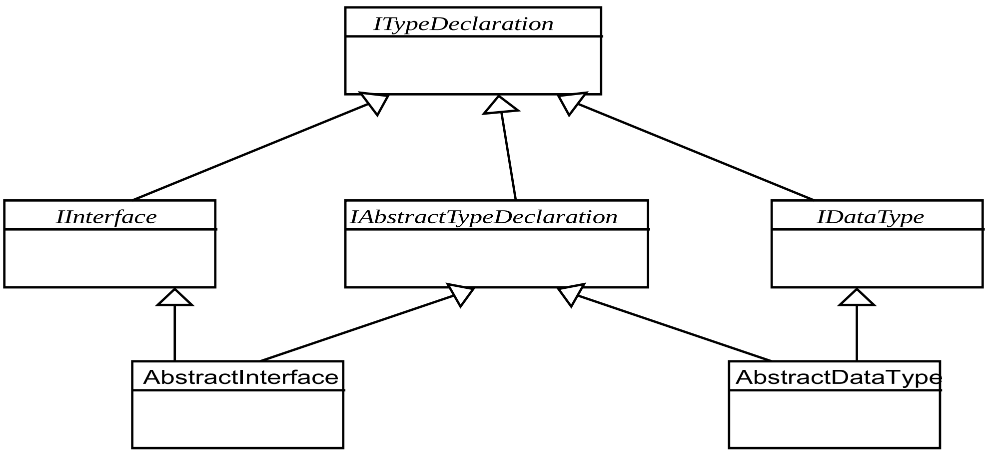

Figure 19: UCM abstract types

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 may be associated with business elements (components, interfaces, methods, etc.).

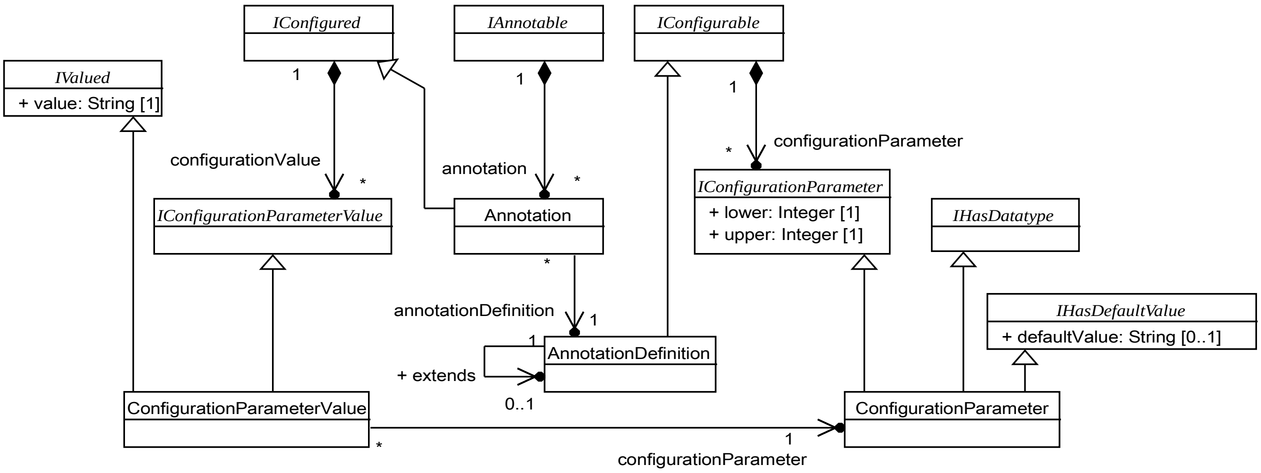

Figure 20: UCM configuration elements

9.2.9.1 ConfigurationParameter (IConfigurationParameter, IHasDataType, IHasDefaultValue)

Configuration parameters are comparable to attributes. Attributes are functional elements, and therefore may be manipulated by business code. Configuration parameters are nonfunctional elements: they have no direction, as they are properties associated with platform elements. They should not 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. Although annotations do not apply to platform elements, annotation definitions contain configuration parameters. This is for metamodel factorization.

- extends: AnnotationDefinition [0…1]

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 is like 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 should 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 shall 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 may reference the same interaction pattern.

The main entities defined in the ucm_interaction package are illustrated in figure 21.

9.3.2 Interaction module

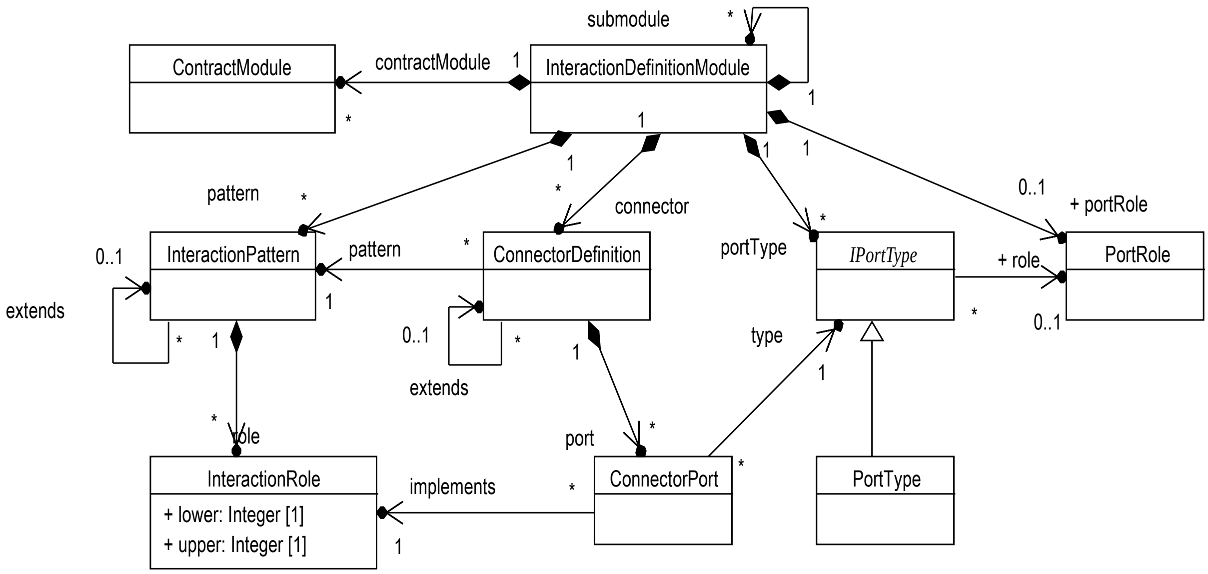

Figure 21: Main classes of the UCM interaction package

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 may be used in a given architecture. An interaction definition module has the following information:

- contractModule: ContractModule [0…*] (owned)
- submodule: InteractionDefinitionModule [0…*] (owned)
- pattern: InteractionPattern [0…*] (owned)
- connector: ConnectorDefinition [0…*] (owned)
- portType: IPortType [0…*] (owned)
- portRole: PortRole [0…*] (owned)

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

9.3.3 Interaction patterns

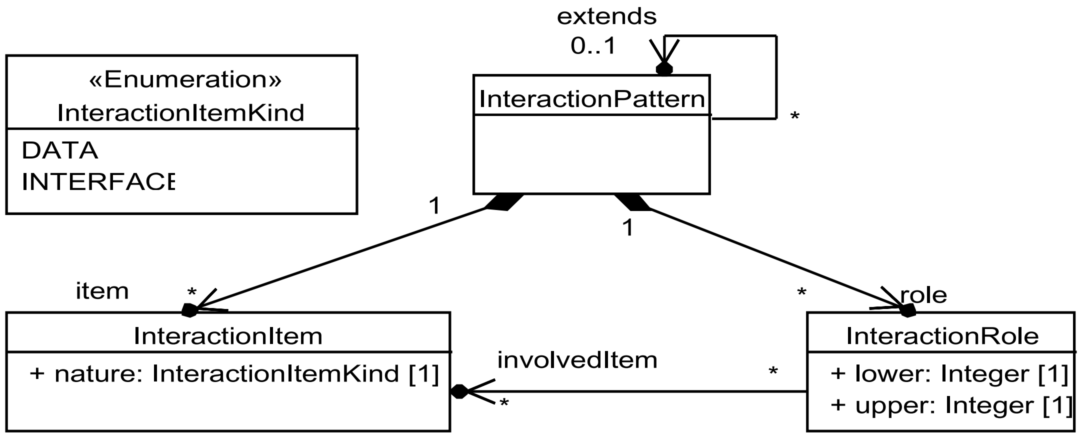

Figure 22: UCM interaction patterns

Interaction patterns provide a definition of the roles different participants shall have in an interaction. These roles do not entail any API; they only provide high-level semantics on which one shall 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 may have a given role;
- An identifier that bears the semantic of that role;
- Interaction items related to this role.

9.3.3.1 InteractionPattern (INamed)

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…*] (owned)
- item: InteractionItem [0…*] (owned)
- extends: InteractionPattern [0…1]

An interaction pattern may extend another interaction pattern to define additional roles. Roles shall not be redefined.

9.3.3.2 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] (owned)

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.

9.3.3.3 InteractionRole (INamed)

- lower: Integer [1] (owned)
- upper: Integer [1] (owned)
- invlovedItem: InteractionItem [0…*]

Fields lower and upper specify how many times the given role may be involved in a given interaction pattern. If lower is set to 0, then the role is optional. The value of field upper shall be greater than or equal to the value of field lower; however, if there is no upper limit, the value of field upper shall be set to -1. The value of field lower shall be greater than or equal to 0. The default values of fields lower and upper is 1. 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.

9.3.4 Connector definitions

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

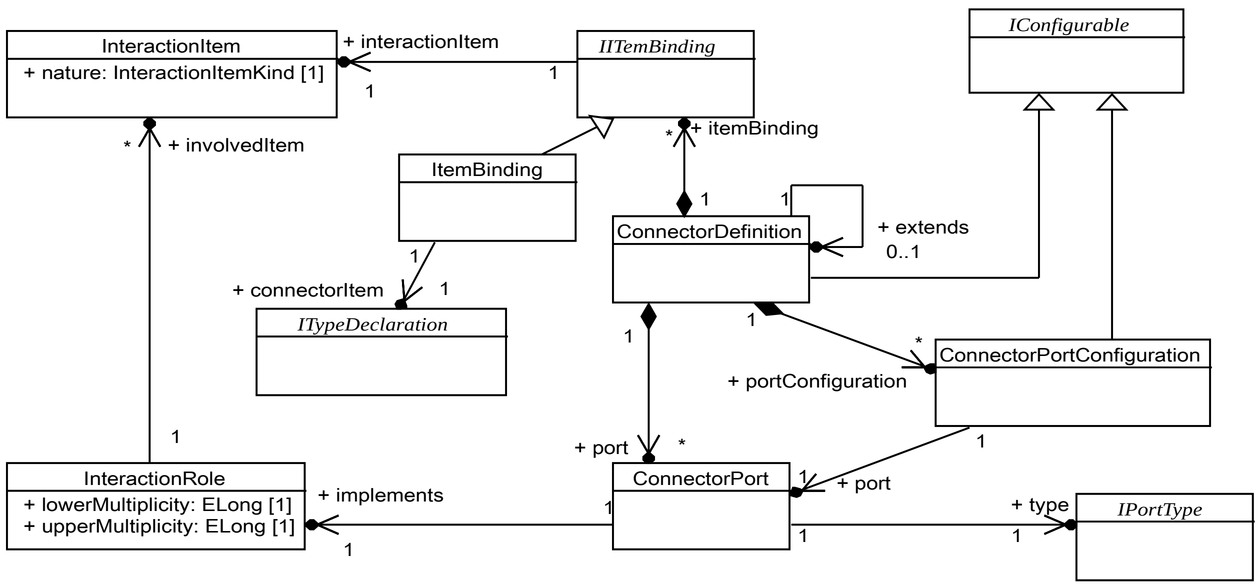

Figure 23: UCM connectors

9.3.4.1 ConnectorDefinition (INamedIConfigurable)

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…*] (owned)
- itemBinding: IItemBinding [0…*] (owned)
- portConfiguration: ConnectorPortConfiguration [0…*] (owned)
- extends: ConnectorDefinition [0…1]

A connector may 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 IItemBinding

Connectors have to specify what interaction items are bound to. This is a way to ensure consistency between the high level specifications of interaction patterns and the detailed APIs of connector definitions. Abstract class IItemBinding and its extension ItemBinding address this need. Extensions of the UCM standard may define other ways of binding interaction items by providing alternative extensions to abstract class IItemBinding.

- interactionItem: InteractionItem [1]

9.3.4.3 ItemBinding (IItemBinding)

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 shall associate all the items of its interaction pattern to data types or interfaces manipulated in its ports. An ItemBinding has the following information:

- connectorItem: ITypeDeclaration [1]

9.3.4.4 ConnectorPort (INamed)

Connector ports correspond to the interaction points of a connector. They define the interaction APIs that are 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.

9.3.4.5 ConnectorPortConfiguration (IConfigurable)

Connector port configurations carry the definitions of the configuration parameters that apply to a given port of the connector definition.

- port: ConnectorPort [1]

The referenced port shall either be a port of the current connector definition or a port of an ancestor connector definition.

9.3.4.6 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 may define other concrete classes to specify APIs.

- role: PortRole [0…1]

9.3.4.7 PortRole (INamed)

Port roles are used to create categories of port types in order to restrict component port refinement (see 9.5.4.2).

9.3.5 Port definitions

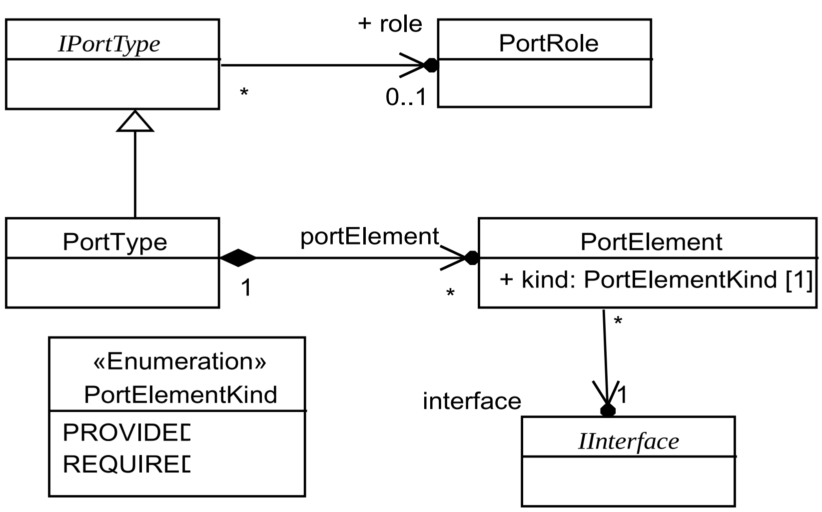

Figure 24: UCM port types

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…*] (owned)

9.3.5.2 PortElement (INamed)

A port element either provides or require an interface.

- interface: IInterface [1]

- kind: PortElementKind [1] (owned)

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

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.

9.4.2 Nonfunctional aspect module

The main entities of the nonfunctional aspects package are illustrated on figure 25. Technical aspects correspond to abstract notions (e.g. component execution policy). Technical policy definitions are the actual means to specify the nonfunctional aspect that are managed by the platform. They may define APIs and may have configuration parameters.

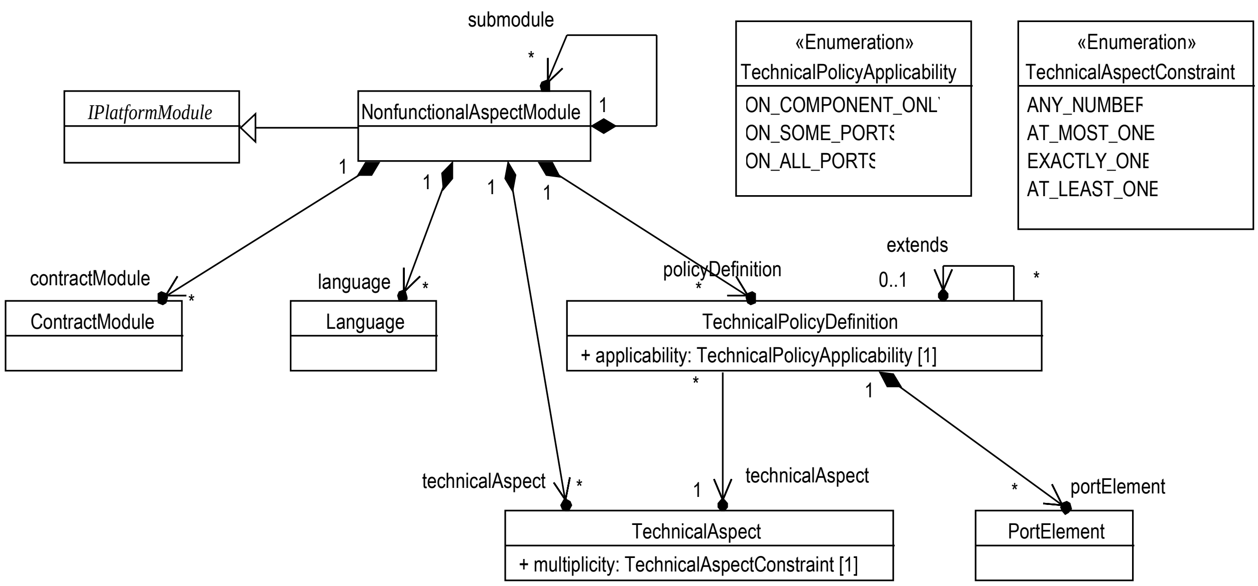

Figure 25: Main classes of UCM technical policies package

9.4.2.1 NonfunctionalAspectModule (IPlatformModule)

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

- policyDefinition: TechnicalPolicyDefinition [0…*] (owned)
- submodule: NonfunctionalAspectModule [0…*] (owned)
- contractModule: ContractModule [0…*] (owned)
- technicalAspect: TechnicalAspect [0…*] (owned)
- languages: Language [0…*] (owned)

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] (owned)

Enumerated type TechnicalAspectConstraint defines the possible multiplicity of technical policies. Four possiblities are defined: ANY_NUMBER, AT_MOST_ONE, EXACTLY_ONE and AT_LEAST_ONE.

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…*] (owned)

- technicalAspect: TechnicalAspect [1]

- applicability: TechnicalPolicyApplicability [1] (owned)

- 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 may have configuration parameters to specify nonfunctional settings (e.g. execution period).

A technical policy definition may 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: ON_COMPONENT_ONLY, ON_SOME_PORTS, ON_ALL_PORTS.

A technical policy shall thus be legally associated with a component, or with one or several ports of a component. Value ON_COMPONENT_ONLY means the technical policy shall not be associated to any port of policy of the component. Value ON_SOME_PORTS means the technical policy shall manage at least one port or policy. Value ON_ALL_PORTS means the technical policy definition is implicitly associated to all the ports and policies of the component. 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.

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 Language (INamed)

Field name 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 may have several implementations. Component implementations may 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.

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 26.

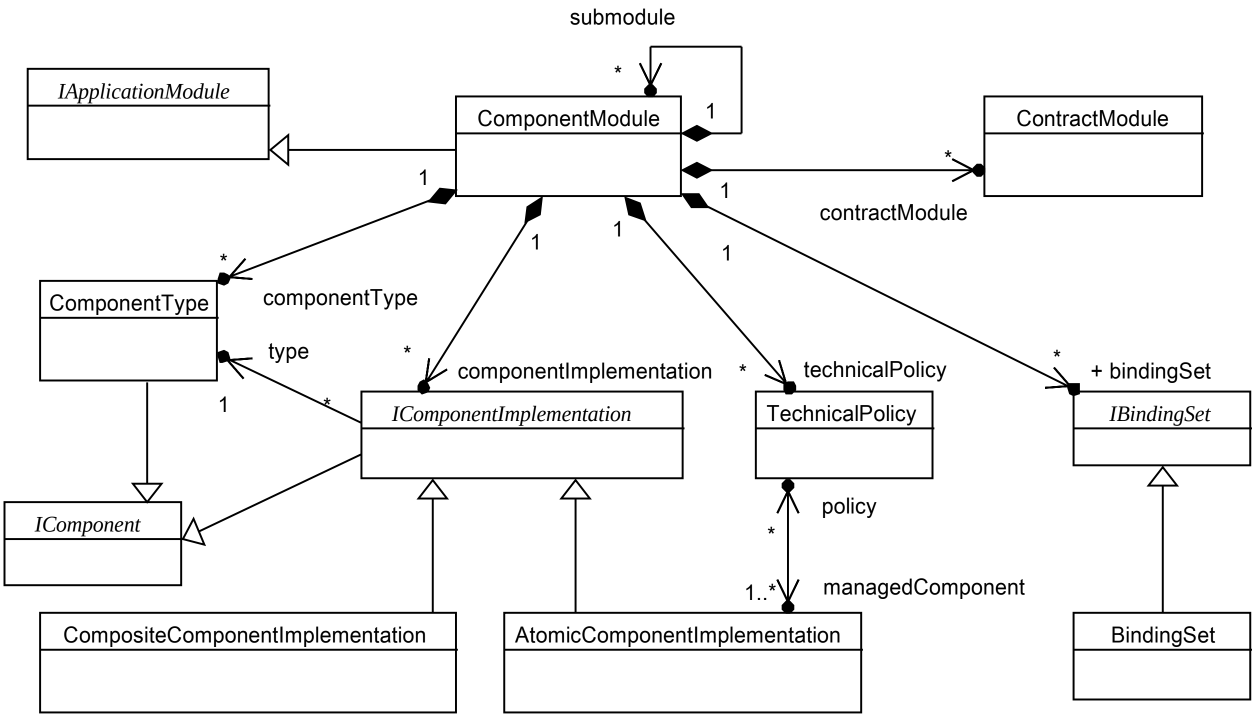

Figure 26: Main classes involved in the UCM component package

9.5.2.1 ComponentModule (IApplicationModule)

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

- submodule: ComponentModule [0…*] (owned)
- contractModule: ContractModule [0…*] (owned)
- componentType: ComponentType [0…*] (owned)
- componentImplementation: IComponentImplementation [0…*] (owned)
- technicalPolicy: ComponentTechnicalPolicy [0…*] (owned)
- bindingSet: IBindingSet [0…*] (owned)

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 may 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 (§ 9.5.5) and CompositeComponentImplementation (§ 9.5.6).

- type: ComponentType [1]

9.5.3 Binding Sets

Binding Sets specify bindings between abstract types (either data or interface) and actual types. A binding set may be referenced in the declaration of a component port or a technical policy if the corresponding definition (port type or technical policy definition) relies on abstract types.

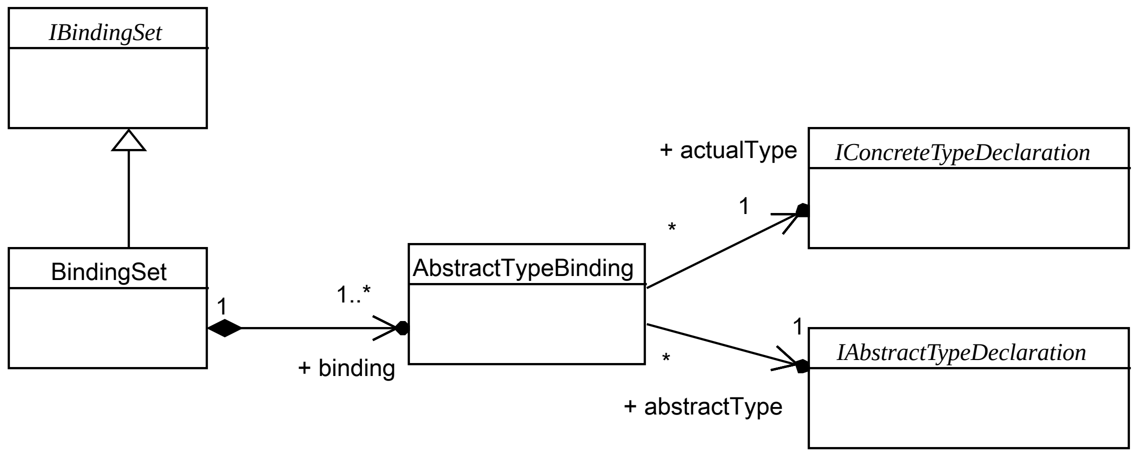

Figure 27: UCM binding sets

9.5.3.1 IBindingSet (INamed)

IBindingSet is an abstract class. Its purpose is to allow the definition of alternative kinds of binding sets in meta-model extensions

9.5.3.2 BindingSet (IBindingSet)

Binding sets gather bindings between abstract types and actual types.

- binding: AbstractTypeBinding [1…*] (owned)

9.5.3.3 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: IConcreteTypeDeclaration [1]

9.5.4 Component types and ports

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

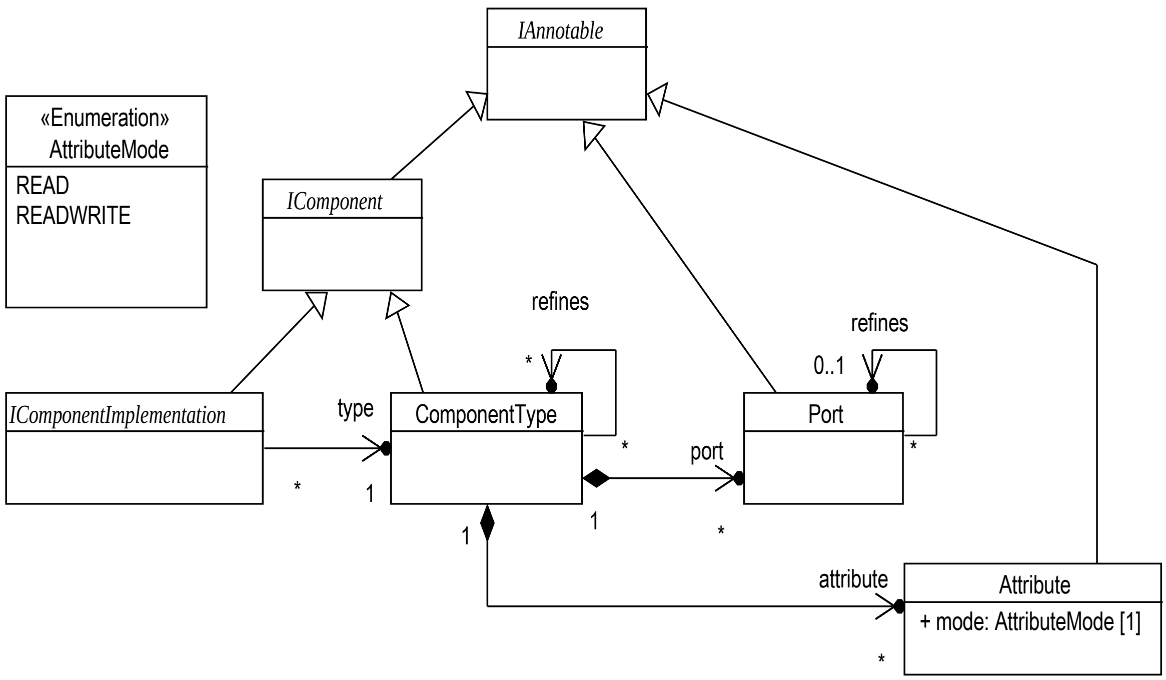

Figure 28: UCM component types

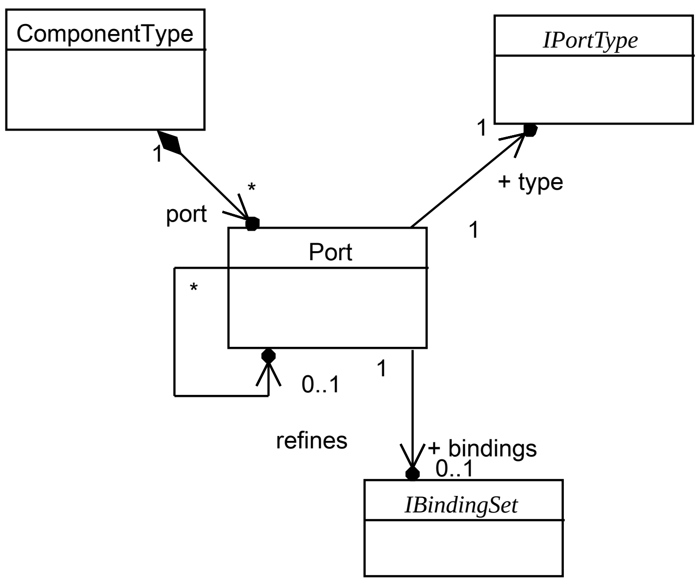

Figure 29: UCM component ports

9.5.4.1 ComponentType (IComponent)

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

- port: Port [0…*] (owned)
- attribute: Attribute [0…*] (owned)
- refines: ComponentType [0…*]

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

A component type may refine other component types. In this situation, the component type inherits the ports and attributes of its ancestors. 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 shall not be used in place of one of its ancestors. The refinement relationship is thus an inheritance relationship but not a subtyping relationship.

In case of component refinement, the descendant component has the union of the ports and attributes of its direct ancestors. In particular, consider the following situation: a component type A has a port pA, a component type B refines A and refines pA into pB (i.e. pB replaces pA), a component type C also refines A, but does not refine any port (and consequently has port pA). Then, if a component type D refines both B and C, it shall have ports pA (from C) and pB (from B).

A port of a given component type shall not have the same name as a port of its ancestor, unless it refines it (§ 9.5.4.2). An attribute shall not have the same name as an attribute of an ancestor of the component.

9.5.4.2 Port (INamed, IAnnotable)

Ports specify component interaction points. They reference a type and possibly a binding set.

- type: IPortType [1]
- refines: Port [0…1]
- bindings: IBindingSet [0…1]

The refinesPort field is used in case of port refinement. The refined port shall be contained in an ancestor component definition. The port types of the refined port and the refining port shall reference the same port role. Consequently, a port, the port type of which references no port role, shall not be refined. It does not need to have the same name as the refining port. A given port may be refined by several ports at a time; it means the port refinement actually leads to decomposition into several ports.

Ports may have annotations. Annotations may be typically used to specify assumptions made by the component in order to execute properly. For example, an annotation may be associated with a component port to indicate an expected rate for data inputs.

9.5.5 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.

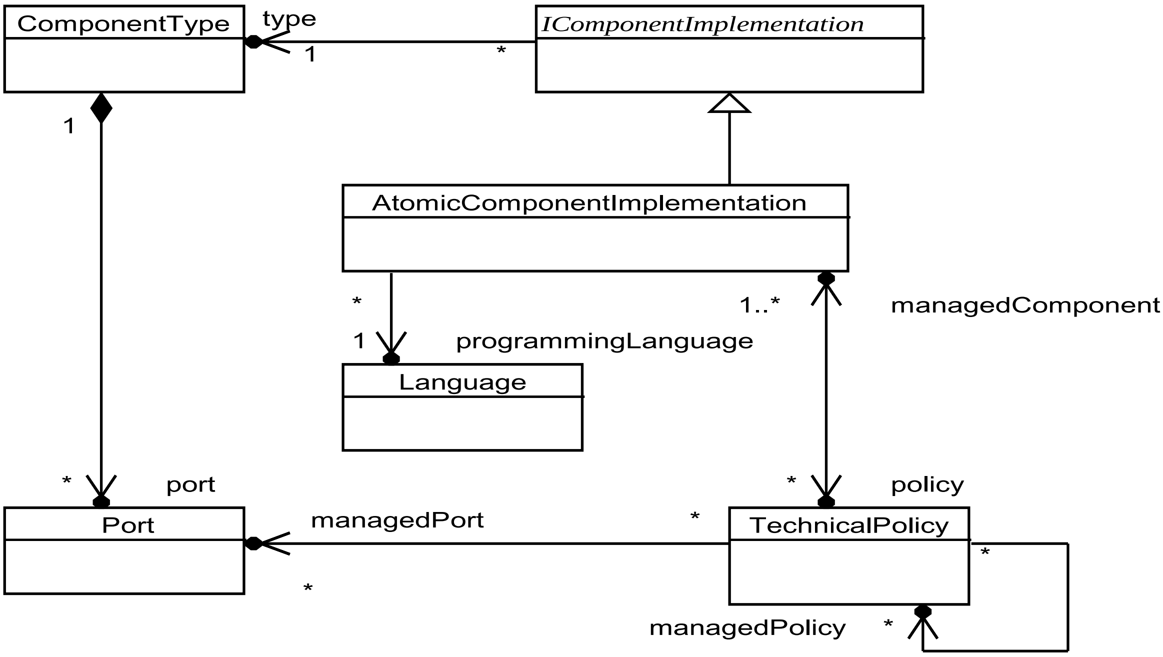

Figure 30: UCM atomic component implementations

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

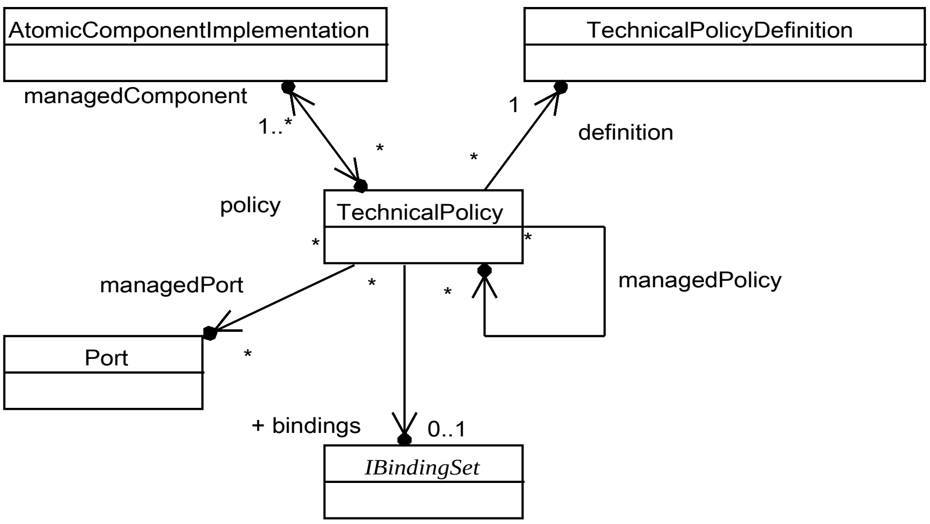

Figure 31: UCM technical policies

9.5.5.1 AtomicComponentImplementation (IComponentImplementation)

Class AtomicComponentImplementation represent actual business logic.

- programmingLanguage: Language [1]
- policy: TechnicalPolicy [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 may be associated with technical policies to specify interactions with or configurations of the component container.

9.5.5.2 TechnicalPolicy (INamed, IConfigured)

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]
- bindings: IBindingSet [0…*]
- managedPort: Port [0...*]
- managedPolicy: TechnicalPolicy [0...*]

Attributes managedPort and managedPolicy are used to specify which interaction points the policy applies to if the applicability set in its definition is ON_SOME_PORTS. If the definition applicability is set to ON_COMPONENT_ONLY or ON_ALL_PORTS, fields managedPort and managedPolicy shall be empty.

A given technical policy may be applied to several atomic component implementations at a time. This is syntactic sugar to avoid the repetitive creation of too many technical policies. It is equivalent to creation of a technical policy for each component. Configuration parameters defined in the corresponding technical policy definition may receive values.

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

9.5.6 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 may 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.

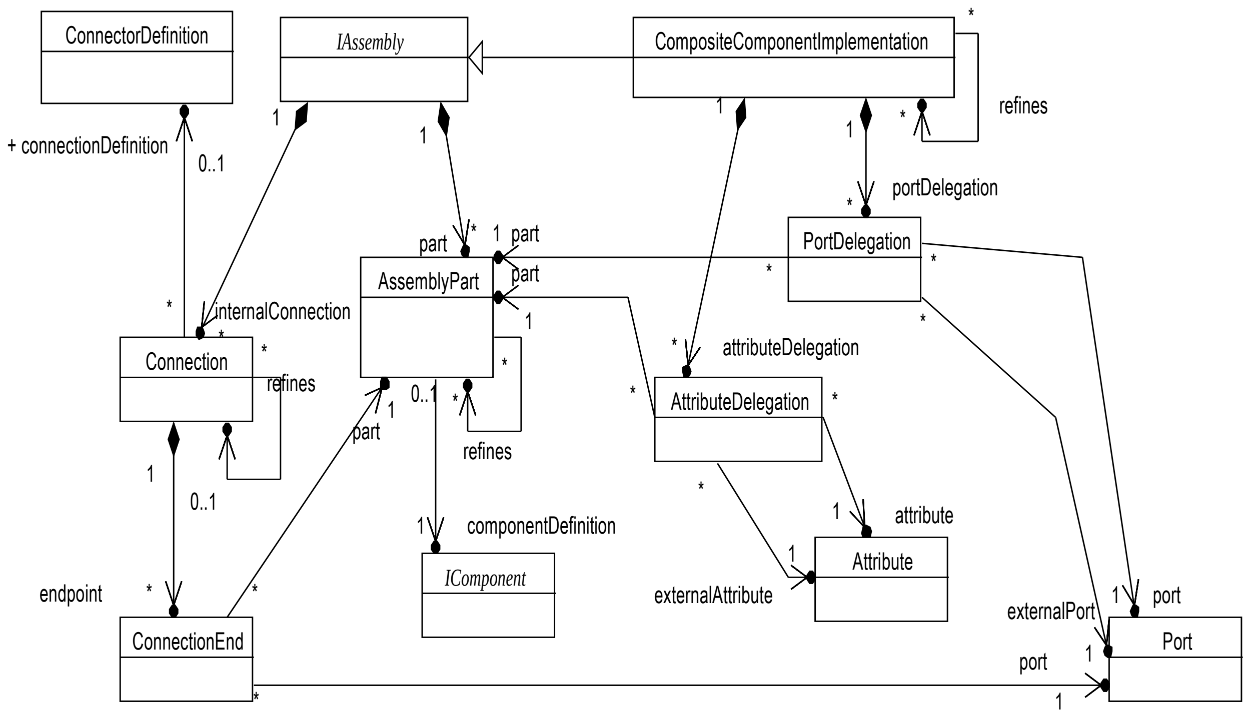

Figure 32: UCM composite component implementations

9.5.6.1 IAssembly

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

- part: AssemblyPart [0…*] (owned)

- internalConnection: Connection [0…*] (owned)

Parts are sub-elements of the assembly.

9.5.6.2 CompositeComponentImplementation (IComponentImplementation, IAssembly)

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

- portDelegation: PortDelegation [0…*] (owned)

- attributeDelegation: AttributeDelegation [0...*] (owned)

- refines: CompositeComponentImplementation [0...*]

A composite component implementation may refine another composite component implementation. In this case, it inherits all the connections, parts and delegations of its ancestor. Additional connections, parts and delegations may be declared.

New declarations may refine inherited ones. The type of the component referenced by a refined part shall be the component type of the inherited part or a refinement of it.

Given a component type CT1 and another component type CT2 that refines CT1, and given two component implementations CT1A and CT1B that implement CT1, and a component implementation CT2A that implements CT2, the following constructions are allowed for a part p2 that would refine a part p1: if p1 references CT1, then p2 may reference CT1A, CT1B, CT2 or CT2A; if p1 references CT1A, then p2 may reference CT1, CT1B, CT2 or CT2A; if p1 references CT2, then p2 may only reference CT2A.

A refined connection may reference any connector definition, provided that the resulting configuration is consistent, i.e. provided that the types of the connected ports are actually managed by the connector definition of the refined connection. It may also declare additional connection ends. If no connector definition is specified, then the refined connection inherits the connector definition of the refined connection.

9.5.6.3 AssemblyPart (INamed, IAnnotable)

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

- componentDefinition: IComponent [1]
- refines: AssemblyPart [0...1]

Assembly parts may 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.

9.5.6.4 Connection (INamed, IConfigured)

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

- endpoint: ConnectionEnd [0…*] (owned)
- connectionDefinition: ConnectorDefinition [0…1]
- refines: Connection [0...1]

Connections may 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.

9.5.6.5 ConnectionEnd (INamed, IConfigured)

Connection ends connect connections to ports of assembly parts.

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

9.5.6.6 PortDelegation (INamed)

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 shall be the same.

- externalPort: Port [1]

- part: AssemblyPart [1]

- port: Port [1]

The external port shall belong to the component type (or one of its ancestors) of the composite component implementation. The part shall reference an assembly part of the composite component implementation, or an assembly part of one of its ancestors. The port shall reference a port of this assembly part (i.e. a port that belongs to the component definition referenced by this part) Unlike connections, port delegations are not associated to a connector definition or an interaction pattern: they simply bind the external port to a port of a sub-component.

9.5.6.7 AttributeDelegation (INamed)

Attribute delegations allow the complete delegation of a component attribute to an attribute of a sub-component. The data type of both attributes shall be the same, but their modes (read only or read/write) may be different.

- externalAttribute: Attribute [1]

- part: AssemblyPart [1]

- attribute: Attribute [1]

The attribute shall belong to the component definition (or to one of its ancestors) referenced by the part. The external attribute shall belong to the component type of the composite component implementation.

An attribute may be delegated to several attributes of several sub-components inside a given composite component implementation.

The semantics of an attribute delegation is the following: if an initial value is set for the attribute of the containing component, this value is actually set for the attribute of the sub-component. If an initial value is also set for the attribute of the sub-component, then it overrides the value set for the attribute of the containing component. This works recursively in case of composites nested in composites.

10. XML syntax for UCM declarations

UCM XML files shall conform to the XML schema for UCM. See ptc/2019-04-06 for the definition of the XML schema. Examples of XML syntax are provided in section 13 and section 17.

10.1 Entity names

Basically, every UCM concept described in the meta-model (see section 9) corresponds to an XML tag. UCM elements are referenced by their name, formatted as follows: ::absolute::module::path::entity, or submodule::entity if the entity is in a submodule. Elements of entities (e.g. component ports) are referenced with a dot, as follows: module::component.port.

10.2 File names

According to the meta-model, a UCM model consists of a module (any class that derives from class IModule) containing declarations, and possibly submodules. Files that contain a UCM model should be named after the top level module. If the top-level module is a name space, then the file should be named after the absolute name of first non-name space module, with module names separated by double hyphens (“--”).

For example, the following declaration should be stored in a file named components.xml:

```
<componentModule name=”components”>
  <compType name=”C1”/>
</componentModule>
```

The following declaration should be stored in a file named global_app--syst1--components.xml:

```
<namespace name=”global_app”>
  <namespace name=”syst1”>
    <componentModule name=”components”>
      <compType name=”C1”/>
    </componentModule>
  </namespace>
</namespace>
```

This page intentionally left blank.

11. Graphical guidelines (non normative)

UCM has no standard graphical syntax; no UML profile has been defined yet. Nevertheless, diagrams are often helpful to understand architectures. This section provides non-normative guidelines to graphically represent UCM elements. The intent is to suggest how to represent illustrative, informal diagrams. The reference syntax is the XML syntax.

11.1 Shapes

UCM defines three main concepts: components, interactions and technical policies. Definitions (component types and implementations, technical aspect, technical policies definitions, interaction patterns, connector definitions) may all be represented by boxes with square corners, with different icons. Components may be represented with a square (■); interactions may be represented with a circle (●); policies may be represented with a diamond (♦). Component ports may be represented by squares, possibly containing an icon that corresponds to the port type or the role they are associated to.

Assembly parts may be represented by boxes with square corners. Connections may be represented by circles or boxes with rounded corners.

Attributes and configuration parameters may be represented by triangles (▲).

Relationships between entities (e.g. definition relationship, extension relationship, etc.) may be represented by lines or arrow; it may be useful to specify the name of the relationship over the lines.

11.2 Colors

Application elements (i.e., component types and implementations) may be represented in blue. Platform elements (interactions and technical policies) may be represented in purple. Contracts (data types, interfaces) may be represented in yellow.

11.3 Example

The following diagram illustrates the graphical guidelines.

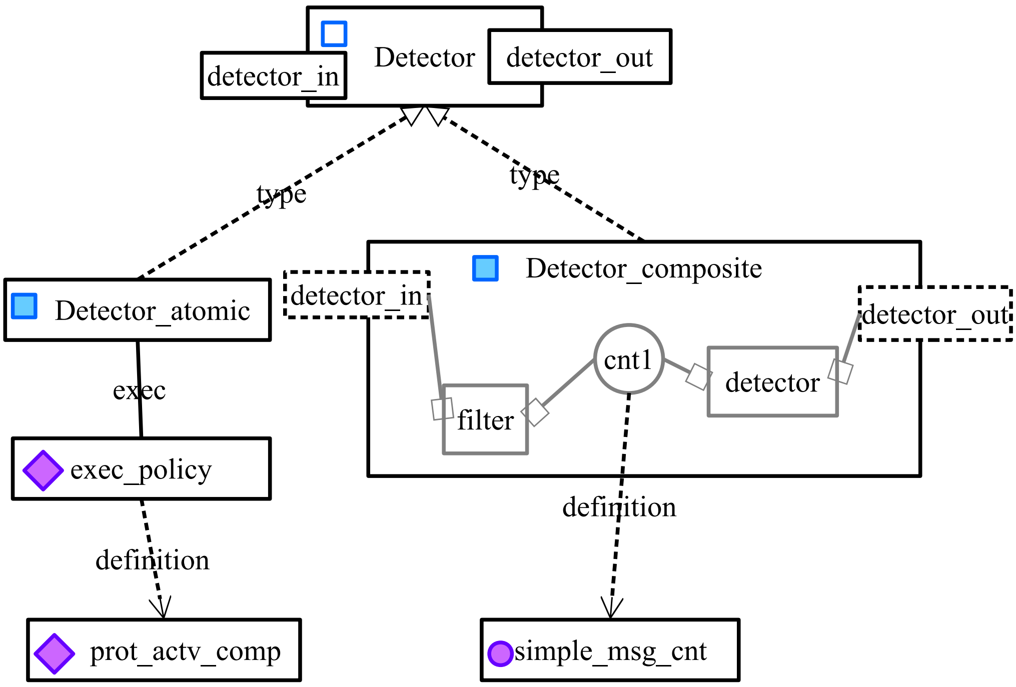

Figure 33: graphical example

It represents a component type named “Detector”, which has two ports: detector_in and detector_out. No information is provided regarding the port types. Component type Detector has two implementations: “Detector_atomic” and “Detector_composite”.

Detector_atomic is linked to a technical policy named “exec_policy” (the same way the two ports are known as “detector_in” and “detector_out” by Detector). The diagrams shows the definition of the technical policy (prot_actv_comp).

Detector_composite has two assembly parts “filter” and “detector”. Their definitions are not represented. Port detector_in is delegated to a port of filter and port detector_out is delegated to a port of detector. Assembly part filter is connected to detector through connection “cnt1”, the definition of which is “simple_msg_cnt”.

12. IDL syntax for UCM declarations

This section explains how to use IDL as a concrete syntax for some of the UCM concepts. IDL cannot represent all UCM concepts, as many of them do not deal with APIs (e.g. composite component implementations).

Although it shares many definitions with it, it must not be confused with section 15 which describes the mapping of the UCM programming model onto the IDL syntax.

12.1 Concerned IDL building blocks

From the IDL separation of the grammar in building blocks, the following blocks are used:

- BB Basic core – Core Data Types
- BB Annotations
- BB Interface – Basic
- BB Components – Basic
- BB Components – Ports and Connectors
- BB Template Modules

12.2 Modules

All kinds of UCM modules (that is, all the classes of the UCM meta-model that derive from class IModule) are represented by IDL modules.

12.3 Contracts

UCM concepts for contracts (interfaces, methods, attributes and data types) are aligned with IDL concepts. See section 15 for the description of equivalences between IDL and UCM data types. There are no anonymous types in UCM. Therefore, the following IDL declaration has no equivalent in UCM:

```
interface intf1 {
  long f(in short a);
};
```

The correct way of defining such an interface in UCM is to first define named types.

```
typedef long long_t;
typedef short short_t;
interface intf1 {
  long_t f(in short_t a);
};
```

Abstract data types and abstract interfaces shall be represented by template parameters.

All UCM interfaces are local IDL interfaces. It is therefore not necessary to specify it.

12.4 Interactions

UCM connector definitions are similar to IDL connectors. UCM connector ports are IDL mirror ports. UCM port types are IDL port types. Interaction patterns cannot be described in IDL.

12.5 Technical policies

UCM technical policies have no equivalent in IDL. Nevertheless, their port elements can be represented in IDL, using a port type and the annotation “@policy”. Technical aspects cannot be described in IDL.

```
@policy
porttype policy_def1 {
  provides intf1 service;
};
```

12.6 Components

Only UCM component types and atomic component implementations can be represented in IDL. The internal structure of composite component implementation is out of the IDL scope.

Atomic component implementations are represented by an IDL component. The component ports shall all be extended ports; neither mirror ports, facets, receptacles, event sinks and event sources are allowed. Policies are ports decorated with the “@policy” annotation.

```
component C1_impl {
  port port_type1 p_in;

  @policy
  port policy_def1 policy1;
};
```

UCM component types are represented by IDL components annotated by “@type”.

13. 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 ptc/2019-04-08.

13.1 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.

13.1.1 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 may 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.

13.1.2 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">
  <enum indexType="short" name="comm_ecode">
    <value index="0" name="ok"/>
    <value index="1" name="internal_error"/>
    <value index="2" name="comm_error"/>
  </enum>
</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).

The equivalent IDL declarations are the following.

```
module return_codes
{
 enum comm_ecode
 {
        OK,
        INTERNAL_ERROR,
        COMM_ERROR
 };
};//end of module return_codes
```

13.1.3 Standard component execution policies

The component execution model is managed by the comp_exec_asp technical aspect. A UCM component shall have exactly one execution model technical policy. The UCM core library defines three technical policies: protected active, protected passive and unprotected passive.

The standard defines another technical aspect, named comp_trig_asp, to specify policies that trigger component execution if need be. A given component may be associated to zero or more triggering policies. The UCM core library defines one policy for this technical aspect: self-executing component.

13.1.3.1 Specifications

The corresponding declarations is shown in XML syntax below.

```
<policyModule name="comp_exec">
  <contractModule name="api">
    <interface name="comp_exec_intf">
      <method name="run"/>
    </interface>
  </contractModule>
  <policyDef applicability="on_component_only" aspect="comp_trig_asp"
name="self_exec_comp">
    <comment>self-executing component</comment>
    <portElement interface="api::comp_exec_intf" kind="provided" name="activation"/>
  </policyDef>
  <policyDef applicability="on_component_only" aspect="comp_exec_asp"
name="unpr_pasv_comp">
    <comment>unprotected passive component</comment>
  </policyDef>
  <policyDef applicability="on_all_ports" aspect="comp_exec_asp" name="prot_pasv_comp">
    <comment>protected passive component</comment>
  </policyDef>
  <policyDef applicability="on_all_ports" aspect="comp_exec_asp" name="prot_actv_comp">
    <comment>protected active component</comment>
  </policyDef>
  <technicalAspect constraint="exactly_one" name="comp_exec_asp"/>
  <technicalAspect constraint="any_number" name="comp_trig_asp"/>
</policyModule>
```

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

13.1.3.2 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 may be reentrant: several components may call it at a time. There is no API. Passive components are like software libraries.

13.1.3.3 Equivalent IDL syntax

Only the self-executing component technical policy defines an API. Therefore only this technical policy has an equivalent in IDL syntax.

```
module comp_exec
 {
  module api
  {
   interface comp_exec_intf
   {
       void run ();
   };
  };//end of module api

  @policy
  porttype self_exec_comp
  {
   provides api::comp_exec_intf activation;
  };
 };//end of module comp_exec
```

13.1.4 Clock and trace service

13.1.4.1 Clock

The core UCM standard defines a technical aspect for a clock service that containers may provide to their components. A UCM component may 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.

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.

13.1.4.2 Trace

The core UCM standard defines a technical aspect for a trace service that containers may provide to their components. A UCM component may 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. The second policy, which applies to ports or policies, should log all calls of all methods of all the port elements. UCM extensions may define alternative trace technical policies.

13.1.4.3 Specifications

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

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

```
<policyModule name="clock">
  <contractModule name="api">
    <struct name="ucm_timeval_t">
      <comment>inspired from the libC definitions</comment>
      <field name="utv_sec" type="ucm_time_t"/>
      <field name="utv_usec" type="ucm_usecond_t"/>
    </struct>
    <integer kind="uint32" name="ucm_time_t"/>
    <integer kind="int32" name="ucm_usecond_t"/>
    <interface name="clk_intf">
      <method name="get_local_time">
        <param dir="out" name="local_time" type="ucm_timeval_t"/>
      </method>
      <method name="get_synchronized_time">
        <param dir="out" name="synchronized_time" type="ucm_timeval_t"/>
      </method>
    </interface>
  </contractModule>
  <policyDef applicability="on_component_only" aspect="clock_asp" name="clock">
    <portElement interface="api::clk_intf" kind="required" name="clk"/>
  </policyDef>
 <technicalAspect constraint="any_number" name="clock_asp"/>
</policyModule>
```

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.

```
<policyModule name="trace">
  <contractModule name="api">
    <enum indexType="uint32" name="log_severity_t">
      <value index="0" name="TRACE"/>
      <value index="1" name="DEBUG"/>
      <value index="2" name="INFO"/>
      <value index="3" name="WARNING"/>
      <value index="4" name="ERROR"/>
      <value index="5" name="CRITICAL"/>
    </enum>
    <string base="wchar" name="log_message_t"/>
    <interface name="trace_intf">
      <method name="log">
        <param dir="in" name="severity" type="log_severity_t"/>
        <param dir="in" name="message" type="log_message_t"/>
      </method>
    </interface>
  </contractModule>
  <policyDef applicability="on_component_only" aspect="trace_asp" name="comp_trace">
    <portElement interface="api::trace_intf" kind="required" name="trace"/>
  </policyDef>
  <policyDef applicability="on_some_ports" aspect="trace_asp" name="port_trace">
    <configParam name="log_severity" type="api::log_severity_t"/>
  </policyDef>
  <technicalAspect constraint="any_number" name="trace_asp"/>
</policyModule>
```

13.1.4.4 Equivalent IDL syntax

The equivalent IDL declarations for the basic service APIs are as follows.

```
module basic_svc
{
  module clock
  {
    module api
    {
        typedef unsigned long ucm_time_t;
        typedef long ucm_second_t;
        struct ucm_timeval_t
        {
          ucm_time_t utv_sec;
          ucm_usecond_t utv_usec;
        };

        interface clk_intf
        {
          void get_local_time (out local_timeout ucm_timeval_t local_time);
          void get_synchronized_time (out ucm_timeval_t synchronized_time);
        };
    };//end of module api

    @policy
    porttype clock
    {
        uses api::clock_intf clock;
    };
  };//end of module clock

  module trace
  {
    module api
    {
        enum log_severity_t
        {
           TRACE,
           DEBUG,
           INFO,
           WARNING,
           ERROR,
           CRITICAL
        };

        typedef wstring log_message_t;

        interface trace_intf
        {
          void log (in log_severity_t severity, in log_message_t message);
        };
    };//end of module api

    @policy
    porttype comp_trace
    {
        uses api::trace_intf trace;
    };

  };//end of module trace
};//end of module basic_svc
```

13.1.5 Standard port roles

13.1.5.1 Description

The UCM core library defines four roles: producer, consumer, client and server. They correspond to the most common roles identified in interactions and are referenced by the standard port types. Platform provider should rely on these role definitions if possible when defining additional, non-standard port types, and avoid defining new, non-standard port roles.

Role producer corresponds to data emission. Role consumer corresponds to data reception. Role client correspond to data emission then reception, or service request. Role server corresponds to data reception then emission, or service provision.

13.1.5.2 Specifications

```
<interactionModule name="roles">
  <portRole name="producer"/>
  <portRole name="consumer"/>
  <portRole name="client"/>
  <portRole name="server"/>
</interactionModule>
```

13.1.6 Service based interaction

13.1.6.1 Description

Service interaction correspond to the classical client / server interaction. It involves two roles: a client and a server. There may be several clients, and there shall be a unique server.

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.

13.1.6.2 Specifications

```
<interactionModule name="services">
  <contractModule name="api">
    <abstractInterface name="service_intf_t"/>
  </contractModule>
  <pattern name="svc_intr_pat">
    <item name="service_item" nature="interface"/>
    <role max="-1" name="client">
      <itemRef ref="service_item"/>
    </role>
    <role name="server">
      <itemRef ref="service_item"/>
    </role>
  </pattern>
  <portType name="svc_srvr_pt" role="::ucm_core::roles::server">
    <portElement interface="api::service_intf_t" kind="provided" name="srvr_pe"/>
  </portType>
  <portType name="svc_cli_pt" role="::ucm_core::roles::client">
    <portElement interface="api::service_intf_t" kind="required" name="cli_pe"/>
```

```
  </portType>
  <connectorDef name="simple_svc_cnt" pattern="svc_intr_pat">
    <connPort name="client" role="client" type="svc_cli_pt"/>
    <connPort name="server" role="server" type="svc_srvr_pt"/>
    <itemBinding cItem="api::service_intf_t" pItem="service_item"/>
  </connectorDef>
</interactionModule>
```

13.1.6.3 Equivalent IDL syntax

The equivalent IDL declarations for the service connector are as follows.

```
module services < interface SERVICE_INTF_T >
{
 porttype svc_srvr_pt
 {
  provides SERVICE_INTF_T srvr_pe;
 };

 porttype svc_cli_pt
 {
  uses SERVICE_INTF_T cli_pe;
 };

 connector simple_svc_cnt
 {
  mirrorport svc_cli_pt client;
  mirrorport svc_srvr_pt server;
 };
}
```

13.1.7 Message based interaction

13.1.7.1 Description

The UCM message base interaction is inspired by CCM message ports. The interaction pattern involves two roles: an emitter and a receiver. There may 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 type named “msg_emtr_pt”. This port type contains a single port element that requires interface “message_intf”. The receiver port references a port type named “msg_rcvr_pt”. This port type also contains a single port element the provides the same interface.

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”.

13.1.7.2 Specifications

```
<interactionModule name="messages">
  <contractModule name="api">
    <abstractDataType name="transm_data_t"/>
    <interface name="msg_emtr_intf">
      <method name="push">
        <param dir="in" name="message" type="transm_data_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
    <interface name="msg_rcvr_intf">
      <method name="push">
```

```
        <param dir="in" name="message" type="transm_data_t"/>
      </method>
    </interface>
  </contractModule>
  <pattern name="msg_intr_pat">
    <item name="message_item" nature="data"/>
    <role max="-1" name="emitter">
      <itemRef ref="message_item"/>
    </role>
    <role max="-1" name="receiver">
      <itemRef ref="message_item"/>
    </role>
  </pattern>
  <portType name="msg_emtr_pt" role="::ucm_core::roles::producer">
    <portElement interface="api::msg_emtr_intf" kind="required" name="emtr_pe"/>
  </portType>
  <portType name="msg_rcvr_pt" role="::ucm_core::roles::consumer">
    <portElement interface="api::msg_rcvr_intf" kind="provided" name="rcvr_pe"/>
  </portType>
  <connectorDef name="simple_msg_cnt" pattern="msg_intr_pat">
    <connPort name="emitter" role="emitter" type="msg_emtr_pt"/>
    <connPort name="receiver" role="receiver" type="msg_rcvr_pt"/>
    <itemBinding cItem="api::transm_data_t" pItem="message_item"/>
  </connectorDef>
</interactionModule>
```

13.1.7.3 Equivalent IDL syntax

The equivalent IDL declarations for the message connector are the following.

```
module messages < typename MESSAGE_TYPE_T >
{
  module api
  {
    interface message_emtr_intf
    {
      ::ucm_core::return_codes::comm_ecode push (in MESSAGE_TYPE_T message);
    };
    interface message_rcvr_intf
    {
      void push (in MESSAGE_TYPE_T message);
    };
  }

  porttype msg_emtr_pt
  {
    uses api::message_emtr_intf emtr_pe;
  };

  porttype msg_rcvr_pt
  {
    provides api::message_rcvr_int rcvr_pe;
  };

  connector simple_msg_cnt
  {
    mirrorport msg_emtr_pt emitter;
    mirrorport msg_rcvr_pt emitter;
```

```
  };
}
```

13.2 Standard properties (Normative, not mandatory)

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

```
<contractModule name="ucm_std_ann">
  <string base="wchar" name="prop_str_t"/>
  <annotationDef name="comp_descr">
    <comment>This annotation may apply to components</comment>
    <configParam name="description" type="prop_str_t"/>
    <configParam name="category" type="prop_str_t"/>
    <configParam name="version" type="prop_str_t"/>
    <configParam name="vendor" type="prop_str_t"/>
  </annotationDef>
</contractModule>
```

13.3 Advanced timer service (Normative, not mandatory)

The component execution policies defined in the core platform specifications (section 13.1.3) allow the definition of self-executing components: the business code of these components shall 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.

13.3.1 Object-based timers

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

The definition of the technical policy and associated contracts is specified by the following declarations.

```
<policyModule name="ott_timer">
  <contractModule name="api">
    <comment>UCM object-oriented timed trigger contract</comment>
    <integer kind="uint32" name="ott_round_t"/>
    <string base="char8" name="ott_str_id"/>
    <bool name="ott_bool_t"/>
    <interface name="ott_handler">
      <method name="on_trigger">
        <param dir="in" name="timer" type="ott_service_intf"/>
        <param dir="in" name="delta_time"
type="::ucm_core::basic_svc::clock::api::ucm_timeval_t"/>
        <param dir="in" name="round" type="ott_round_t"/>
      </method>
    </interface>
    <interface name="ott_service_intf">
      <attribute mode="read" name="rounds" type="ott_round_t"/>
      <attribute mode="read" name="id" type="ott_str_id"/>
      <method name="cancel"/>
      <method name="is_cancelled">
        <param dir="return" name="returns" type="ott_bool_t"/>
      </method>
    </interface>
```

```
    <interface name="ott_scheduler">
      <method name="scheduler_trigger">
        <param dir="return" name="returns" type="ott_service_intf"/>
        <param dir="in" name="trigger_handler" type="ott_handler"/>
        <param dir="in" name="trigger_delay"
type="::ucm_core::basic_svc::clock::api::ucm_timeval_t"/>
      </method>
      <method name="schedule_repeated_trigger">
        <param dir="in" name="trigger_handler" type="ott_handler"/>
        <param dir="in" name="start_delay"
type="::ucm_core::basic_svc::clock::api::ucm_timeval_t"/>
        <param dir="in" name="interval"
type="::ucm_core::basic_svc::clock::api::ucm_timeval_t"/>
        <param dir="in" name="max_rounds" type="ott_round_t"/>
      </method>
    </interface>
  </contractModule>
  <policyDef applicability="on_component_only"
aspect="::ucm_core::comp_exec::comp_trig_asp" name="ott_timer">
    <portElement interface="api::ott_scheduler" kind="required" name="timer_scheduler"/>
  </policyDef>
</policyModule>
```

The equivalent IDL declarations are the following.

```
module ott_timer
 {
   module api
   {
    typedef unsigned long ott_round_t;
    typedef char ott_str_id;
    typedef boolean ott_bool_t;

    interface ott_service_intf
    {
         readonly attribute ott_round_t rounds;
         readonly attribute ott_str_id id;
         void cancel ();
         ott_bool_t is_canceled ();
    };

    interface ott_handler
    {
         void on_trigger (in ott_service_intf timer,
in ::ucm_core::basic_svc::clock::api::ucm_timeval_t delta_time, in ott_round_t round);
    };

    interface ott_scheduler
    {
         ott_service_intf scheduler_trigger (in ott_handler trigger_handler,
                                         in ::ucm_core::basic_svc::clock::api::ucm_timeval_t trigger_delay);
         void schedule_repeated_trigger (in ott_handler trigger_handler,
                                              in ::ucm_core::basic_svc::clock::api::ucm_timeval_t
start_delay,
                                              in ::ucm_core::basic_svc::clock::api::ucm_timeval_t interval,
                                              in ott_round_t max_rounds);
    };

   };//end of module api
```

```
@policy
porttype ott_timer
{
  uses api::ott_scheduler timer_scheduler;
};

};//end of module ott_timer
```

13.3.2 Index-based timers

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

The definition of the technical policy and associated contracts is specified by the following declarations.

```
<policyModule name="itt_timer">
  <contractModule name="api">
    <comment>UCM id-based timed trigger contract</comment>
    <struct name="timeout_t">
      <field name="time_val" type="::ucm_core::basic_svc::clock::api::ucm_timeval_t"/>
      <field name="flag" type="timeout_enum_t"/>
    </struct>
    <enum indexType="int16" name="timeout_enum_t">
      <value index="0" name="ABSOLUTE_TIME"/>
      <value index="1" name="RELATIVE_TIME"/>
    </enum>
    <integer kind="uint32" name="timer_id_t"/>
    <bool name="timer_bool_t"/>
    <interface name="itt_callback_intf">
      <method name="on_timeout">
        <param dir="in" name="time" type="timeout_t"/>
        <param dir="in" name="timer_number" type="timer_id_t"/>
      </method>
    </interface>
    <interface name="itt_service_intf">
      <method name="start_periodic_scheduler">
        <param dir="in" name="timer_id" type="timer_id_t"/>
        <param dir="in" name="delay_time" type="timeout_t"/>
        <param dir="in" name="rate" type="timeout_t"/>
      </method>
      <method name="start_sporadic_scheduler">
        <param dir="in" name="timer_id" type="timer_id_t"/>
        <param dir="in" name="time" type="timeout_t"/>
      </method>
      <method name="cancel_timer">
        <param dir="in" name="timer_id" type="timer_id_t"/>
      </method>
      <method name="is_canceled">
        <param dir="in" name="timer_id" type="timer_id_t"/>
        <param dir="return" name="returns" type="timer_bool_t"/>
      </method>
    </interface>
  </contractModule>
  <policyDef applicability="on_component_only"
aspect="::ucm_core::comp_exec::comp_trig_asp" name="itt_timer">
    <portElement interface="api::itt_callback_intf" kind="provided"
name="timer_callback"/>
```

```
    <portElement interface="api::itt_service_intf" kind="required" name="timer_service"/>
  </policyDef>
</policyModule>
```

The equivalent IDL declarations are the following.

```
module itt_timer
 {
   module api
   {
     enum timeout_enum_t
     {
     ABSOLUTE_TIME,
     RELATIVE_TIME
     };

     typedef short timer_id_t;

     typedef boolean timer_bool_t;

     struct timeout_t
     {
     ::ucm_core::basic_svc::clock::api::ucm_timeval_t time_val;
     timeout_enum_t flag;
     };

     interface itt_callback_intf
     {
     void on_timeout (in timeout_t time, in timer_id_t timer_number);
     };

     interface itt_service_intf
     {
     void start_periodic_scheduler (in timer_id_t timer_id,
                                    in timeout_t delay_time,
                                    in timeout_t rate);
     void start_sporadic_scheduler (in timer_id_t timer_id,
                                    in timeout_t timer);
     void cancel_timer (in timer_id_t timer_id);
     timer_bool_t is_canceled (in timer_id_t timer_id);
     };
   };//end of module api

   @policy
   porttype itt_timer
   {
     provides api::itt_callback_intf timer_callback;
     uses api::itt_service_intf timer_service;
   };
 };//end of module itt_timer
```

Technical policy “itt_timer” has two port elements: one (“timer_callback”) is provided by the component executor, and shall be implemented by the business code. It has a unique method “on_timeout”, which shall 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 shall repeat infinitely; a sporadic timer shall trigger once. Upon the initiation of a timer, the business code shall provide a timer number. Thus, a single timer service may manage several timers, all being associated with the same callback method.

Timers can be canceled.

13.4 Additional interactions (Normative, not mandatory)

The core specifications defines APIs for service and message interactions (sections 13.1.6 and 13.1.7). 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.

13.4.1 Request-response

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

13.4.1.1 Specifications

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.

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.

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

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 shall only be a unique server, either the synchronous server port or the asynchronous server port shall be connected.

```
<interactionModule name="request_response">
  <contractModule name="api">
    <abstractDataType name="req_data_t"/>
    <abstractDataType name="resp_data_t"/>
    <integer kind="uint32" name="rr_id_t"/>
    <interface name="rrsync_cli_intf">
      <comment>interface for request-response synchronous client</comment>
      <method name="request">
        <param dir="in" name="request" type="req_data_t"/>
        <param dir="out" name="response" type="resp_data_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
    <interface name="rrsync_srv_intf">
      <comment>interface for request-response synchronous server</comment>
      <method name="request">
        <param dir="in" name="request" type="req_data_t"/>
        <param dir="out" name="response" type="resp_data_t"/>
      </method>
    </interface>
    <interface name="rrasync_req_srv_intf">
      <comment>interface for request-response asynchronous server (request)</comment>
      <method name="request">
        <param dir="in" name="request" type="req_data_t"/>
        <param dir="in" name="req_id" type="rr_id_t"/>
      </method>
    </interface>
    <interface name="rrasync_resp_cli_intf">
      <comment>interface for request-response asynchronous client (response)</comment>
      <method name="response">
        <param dir="in" name="response" type="resp_data_t"/>
```

```
        <param dir="in" name="resp_id" type="rr_id_t"/>
      </method>
    </interface>
    <interface name="rrasync_resp_srv_intf">
      <comment>interface for request-response asynchronous server (response)</comment>
      <method name="response">
        <param dir="in" name="response" type="resp_data_t"/>
        <param dir="in" name="resp_id" type="rr_id_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
    <interface name="rrasync_req_cli_intf">
      <comment>interface for request-response asynchronous client (request)</comment>
      <method name="request">
        <param dir="in" name="request" type="req_data_t"/>
        <param dir="out" name="req_id" type="rr_id_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
  </contractModule>
  <pattern name="rr_intr_pat">
    <item name="req_data" nature="data"/>
    <item name="resp_data" nature="data"/>
    <role max="-1" name="rr_client">
      <itemRef ref="req_data"/>
      <itemRef ref="resp_data"/>
    </role>
    <role name="rr_server">
      <itemRef ref="req_data"/>
      <itemRef ref="resp_data"/>
    </role>
  </pattern>
  <portType name="rrs_cli_pt" role="::ucm_core::roles::client">
    <portElement interface="api::rrsync_cli_intf" kind="required" name="cli_pe"/>
  </portType>
  <portType name="rrs_srvr_pt" role="::ucm_core::roles::server">
    <portElement interface="api::rrsync_srv_intf" kind="provided" name="srvr_pe"/>
  </portType>
  <portType name="rra_cli_pt" role="::ucm_core::roles::client">
    <portElement interface="api::rrasync_req_cli_intf" kind="required" name="cli_req_pe"/
>
    <portElement interface="api::rrasync_resp_cli_intf" kind="provided"
name="cli_resp_pe"/>
  </portType>
  <portType name="rra_srvr_pt" role="::ucm_core::roles::server">
    <portElement interface="api::rrasync_req_srv_intf" kind="provided"
name="srvr_req_pe"/>
    <portElement interface="api::rrasync_resp_srv_intf" kind="required"
name="srvr_resp_pe"/>
  </portType>
  <connectorDef name="req_resp_cnt" pattern="rr_intr_pat">
    <connPort name="rr_sync_cli" role="rr_client" type="rrs_cli_pt"/>
    <connPort name="rr_sync_srvr" role="rr_server" type="rrs_srvr_pt"/>
    <connPort name="rr_async_cli" role="rr_client" type="rra_cli_pt"/>
    <connPort name="rr_async_srvr" role="rr_server" type="rra_srvr_pt"/>
    <itemBinding cItem="api::req_data_t" pItem="req_data"/>
    <itemBinding cItem="api::resp_data_t" pItem="resp_data"/>
  </connectorDef>
</interactionModule>
```

13.4.1.2 Semantics

Synchronous client and server ports have the same execution semantics as in the service connector (§ 13.1.6): 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 may be stored to be processed later; the response API may 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 may receive several requests before sending responses.

13.4.1.3 Equivalent IDL syntax

The equivalent IDL declarations are the following.

```
module request_response<typename REQ_DATA_T, typename RESP_DATA_T>
{
  module api
  {
    typedef unsigned long rr_id_t;
    interface rrsync_cli_intf
    {
        ::ucm_core::return_codes::comm_ecode request (in REQ_DATA_T request,
                                                      out RESP_DATA_T response);
    };
    interface rrsync_srv_intf
    {
        void request (in REQ_DATA_T request,
                      out RESP_DATA_T response);
    };

    interface rrasync_req_srv_intf
    {
        void request (in REQ_DATA_T request,
                                                      in rr_id_t req_id);
    };

    interface rrasync_resp_srv_intf
    {
        ::ucm_core::return_codes::comm_ecode response (in RESP_DATA_T response,
                                                       in rr_id_t req_id);
    };

    interface rrasync_resp_cli_intf
    {
        void response (in RESP_DATA_T response,
                                                       in rr_id_t req_id);
    };

    interface rrasync_req_cli_intf
    {
        ::ucm_core::return_codes::comm_ecode request (in REQ_DATA_T request,
                                                      out rr_id_t req_id);
    };
  };//end of module api

  porttype rrs_cli_pt
  {
    uses api::rrsync_cli_intf cli_pe;
  };
```

```
porttype rrs_srvr_pt
{
  provides api::rrsync_srv_intf srvr_pe;
};

porttype rra_cli_pt
{
  uses api::rrasync_req_cli_intf cli_req_pe;
  provides api::rrasync_resp_cli_intf cli_resp_pe;
};

porttype rra_srvr_pt
{
  provides api::rrasync_req_srv_intf srvr_req_pe;
  uses api::rrasync_resp_srv_intf srvr_resp_pe;
};

connector req_resp_cnt
{
  mirrorport rrs_cli_pt rr_sync_cli;
  mirrorport rrs_srvr_pt rr_sync_srvr;
  mirrorport rra_cli_pt rr_async_cli;
  mirrorport rra_srvr_pt rr_async_srvr;
};
};//end of module request_response
```

13.4.2 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 13.1.7), 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.

13.4.2.1 Specifications

The shared data interaction is defined in an interaction module.

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.

```
<interactionModule name="shared_data">
  <contractModule name="api">
    <interface name="data_reader">
      <method name="freeze_data">
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
      <method name="release_data">
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
      <method name="read_data">
        <param dir="out" name="data" type="::ucm_core::messages::api::transm_data_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
    <interface name="data_notification">
      <method name="on_data_update">
```

```
        <comment>no error code for this method, since it is called by the
connector</comment>
      </method>
    </interface>
    <interface name="data_writer">
      <method name="write_data">
        <param dir="in" name="data" type="::ucm_core::messages::api::transm_data_t"/>
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
      <method name="publish_data">
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
      <method name="cancel_data">
        <param dir="return" name="ecode" type="::ucm_core::return_codes::comm_ecode"/>
      </method>
    </interface>
  </contractModule>
  <pattern name="sd_intr_pat">
    <item name="shr_data" nature="data"/>
    <role max="-1" name="data_writer">
      <itemRef ref="shr_data"/>
    </role>
    <role max="-1" name="data_reader">
      <itemRef ref="shr_data"/>
    </role>
  </pattern>
  <portType name="sd_writer_pt" role="::ucm_core::roles::producer">
    <portElement interface="api::data_writer" kind="required" name="wrtr_pe"/>
  </portType>
  <portType name="sd_reader_pt" role="::ucm_core::roles::consumer">
    <portElement interface="api::data_reader" kind="required" name="rdr_pe"/>
    <portElement interface="api::data_notification" kind="provided" name="notif_pe"/>
  </portType>
  <connectorDef name="sd_cnt" pattern="sd_intr_pat">
    <connPort name="sd_reader" role="data_reader" type="sd_reader_pt"/>
    <connPort name="sd_writer" role="data_writer" type="sd_writer_pt"/>
    <connPort name="sd_simple_writer" role="data_writer"
              type="::ucm_core::messages::msg_emtr_pt"/>
    <itemBinding cItem="::ucm_core::messages::api::transm_data_t" pItem="shr_data"/>
  </connectorDef>
</interactionModule>
```

13.4.2.2 Semantics

The reader API has two port elements: rdr_pe to fetch data, and notif_pe 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.

There are two possible ports for data writing: sd_writer and sd_simple_writer.

The sd_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.

The sd_simple_writer API is the message send port type. It provides a single method push(). This method is equivalent to calling write_data() then publish_data(). This enables flexibility when specifying connections for components that send messages: a component port with port type msg_emtr_pt can either interact by sending messages or writing data, depending on which connector is actually chosen for the connection.

13.4.2.3 Equivalent IDL syntax

The equivalent IDL declarations are the following.

```
module shared_data<typename SHR_DATA_T>
{
  module api
  {
   interface data_reader
   {
        ::ucm_core::return_codes::comm_ecode freeze_data ();
        ::ucm_core::return_codes::comm_ecode release_data ();
        ::ucm_core::return_codes::comm_ecode read_data (out SHR_DATA_T data);
   };

   interface data_notification
   {
        void on_data_update ();
   };

   interface data_writer
   {
        ::ucm_core::return_codes::comm_ecode write_data (in SHR_DATA_T data);
        ::ucm_core::return_codes::comm_ecode publish_data ();
        ::ucm_core::return_codes::comm_ecode cancel_data ();
   };
  };//end of module api

  porttype sd_writer_pt
  {
   uses api::data_writer wrtr_pe;
  };

  porttype sd_reader_pt
  {
   uses api::data_reader rdr_pr;
   provides api::data_notification notif_pe;
  };

  connector sd_cnt
  {
   mirrorport sd_reader_pt sd_reader;
   mirrorport sd_writer_pt sd_writer;
   mirrorport ::ucm_core::messages::msg_emtr_pt sd_simple_writer;
  };

};//end of module shared_data
```

13.5 Additional component execution policies (Normative, not mandatory)

This section describes extensions to the “protected self-executing component” and “protected active component” technical policies (§ 13.1.3). 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.

13.5.1 Specifications

The two technical policies prdc_self_exec_comp for periodic self execution and background_self-executing_component extend technical policy self-executing_component. They add configuration parameters to specify task priority, etc.

Task priority is used for scheduler configuration and scheduling analysis. Priority 1 is the highest priority. Offset corresponds to the delay between the start of the system and the actual start of the task.

The two technical policies prdc_prot_actv_comp for periodic active protected component and spdc_prot_actv_comp for sporadic protected active component extend the protected active component policy defined in the core library. They add configuration parameters to specify task priority, etc.

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

```
<policyModule name="ucm_ext_exec">
  <contractModule name="contracts">
    <integer kind="uint16" name="priority_t">
      <comment>priority 1 is the highest</comment>
    </integer>
    <enum indexType="uint16" name="ucm_time_unit_t">
      <value index="0" name="s"/>
      <value index="1" name="ms"/>
      <value index="2" name="ns"/>
    </enum>
    <struct name="ucm_duration_t">
      <field name="value" type="ucm_time_value_t"/>
      <field name="unit" type="ucm_time_unit_t"/>
    </struct>
    <integer kind="int32" name="ucm_time_value_t"/>
  </contractModule>
  <policyDef applicability="on_component_only"
aspect="::ucm_core::comp_exec::comp_trig_asp" name="prdc_self_exec_comp">
    <comment>periodic self-executing component. It will invoke method run every period.</
comment>
    <extends ref="::ucm_core::comp_exec::self_exec_comp"/>
    <configParam name="psec_period" type="contracts::ucm_duration_t"/>
    <configParam name="psec_priority" type="contracts::priority_t"/>
    <configParam name="psec_offset" type="contracts::ucm_duration_t"/>
  </policyDef>
  <policyDef applicability="on_component_only"
aspect="::ucm_core::comp_exec::comp_trig_asp" name="bgnd_self_exec_comp">
    <comment>background self-executing component. It will invoke method run only
once.</comment>
    <extends ref="::ucm_core::comp_exec::self_exec_comp"/>
    <configParam name="bsec_priority" type="contracts::priority_t"/>
    <configParam name="bsec_offset" type="contracts::ucm_duration_t"/>
  </policyDef>
  <policyDef applicability="on_all_ports" aspect="::ucm_core::comp_exec::comp_exec_asp"
name="spdc_prot_actv_comp">
    <comment>sporadic protected active component. It will trigger the component execution
```

```
whenever a port (or policy) is triggered.</comment>
    <extends ref="::ucm_core::comp_exec::prot_actv_comp"/>
    <configParam name="spac_priority" type="contracts::priority_t"/>
    <configParam name="spac_min_period" type="contracts::ucm_duration_t"/>
  </policyDef>
  <policyDef applicability="on_all_ports" aspect="::ucm_core::comp_exec::comp_exec_asp"
name="prdc_prot_actv_comp">
    <comment>periodic protected active component. It will trigger the component execution
every period if a port (or policy) is triggered.</comment>
    <extends ref="::ucm_core::comp_exec::prot_actv_comp"/>
    <configParam name="ppac_priority" type="contracts::priority_t"/>
    <configParam name="ppac_period" type="contracts::ucm_duration_t"/>
    <configParam name="ppac_offset" type="contracts::ucm_duration_t"/>
  </policyDef>
</policyModule>
```

13.5.2 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.

13.5.2.1 Equivalent IDL syntax

The technical policies extend the component execution policies and only add configuration parameters. The APIs are the same.

13.6 Programming language declarations (Normative, not mandatory)

In order to enable model and code portability, the standard defines four variations of C++ programming languages. These definitions are variations of C++ mappings (see section 16). Tools that support any of these programming languages shall apply the corresponding standard C++ mapping. Tools may implement alternative C++ mappings or implement mappings to other programming languages; such other mappings shall correspond to other programming language declarations.

```
<namespace name="ucm_lang">
 <policyModule name="cpp">
   <language name="CPP11_generic"/>
   <language name="CPP11_typed"/>
   <language name="CPP03_generic"/>
   <language name="CPP03_typed"/>
 </policyModule>
</namespace>
```

14. UCM container 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.

14.1 Runtime entities

14.1.1 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.

14.1.2 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.

14.1.2.1 From connectors to fragments (not normative)

As stated by the UCM metamodel, a ConnectorDefinition owns a set of ConnectorPort that include, similarly to a component Port, 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 34 depicts an example of that mapping. In that example, each PortElement is realized by a separate fragment.

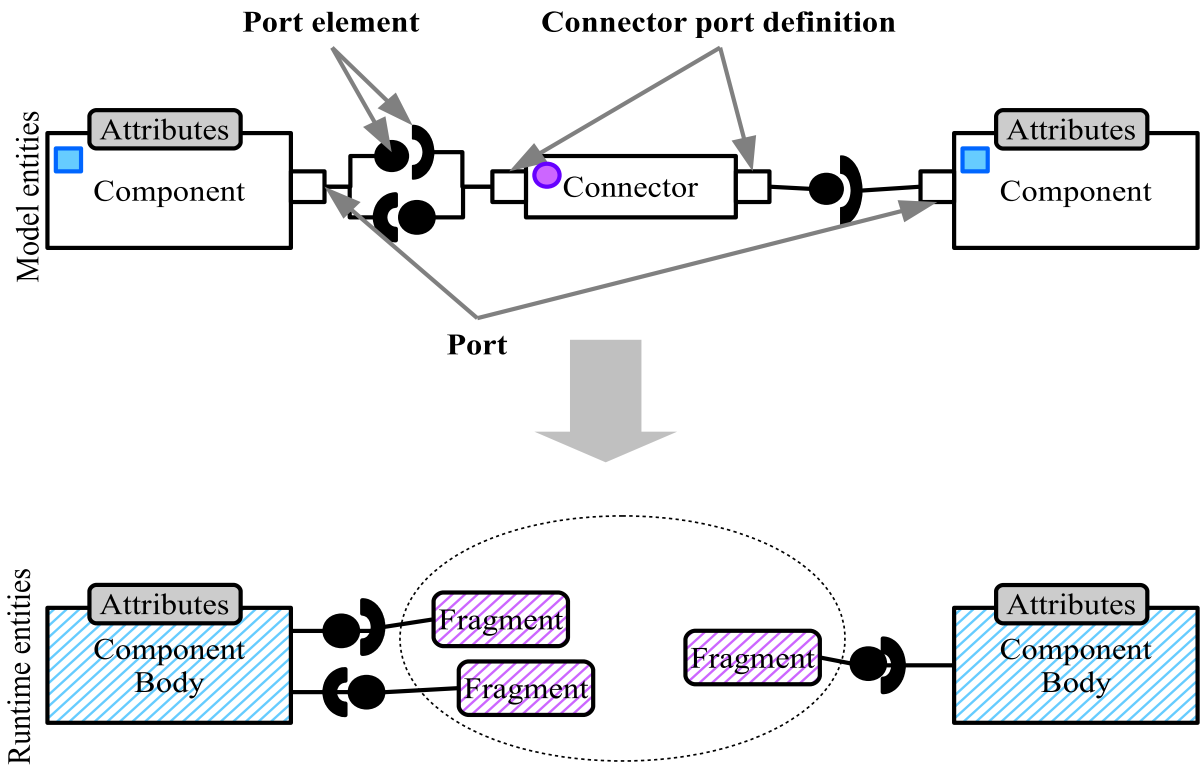

Figure 34: Connector fragmentation example

The communication between the fragments is connector-specific. It is typically based on the communication mechanisms that the connector is intended to abstract. E.g.: the fragments of a 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.

14.1.2.2 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. Like for the connector PortElements, the mapping to fragments is implementation-dependent. Figure 35 shows an example where all the PortElements of the TechnicalPolicy are realized by one fragment.

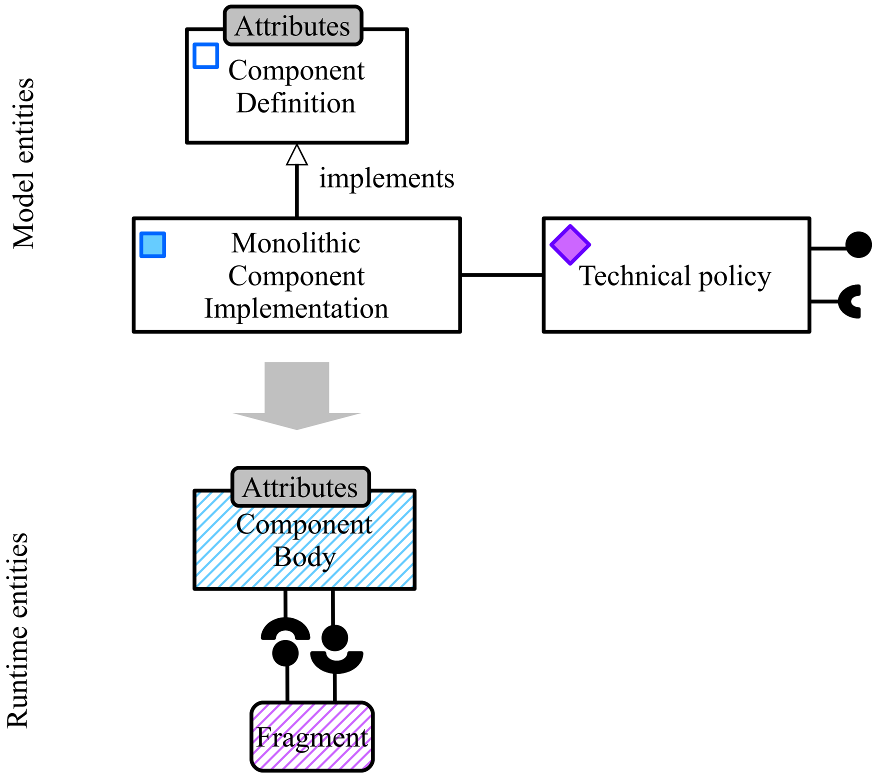

Figure 35: Technical policy fragmentation example

14.1.3 Container

The container is the glue that allows the component body to collaborate with the fragments to make them operational. The main role of the container to manage the component body and fragments life cycle and enable the communication between them.

A container should contain a single component body and all its associated fragments. Containers may be themselves contained in container managers. A container manager is a container that contains sub-containers and possibly a component body and fragments. This allows the creation of container hierarchies in order to ensure consistency in life cycles or to share information between some fragments.

14.2 Container model

The container model defines the different standard interfaces between the different UCM runtime elements, including the container, the component body and the fragments. Figure 36 shows the different interactions that typically 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 model is to be able to implement portable component bodies. That's why all the interactions of the component body with its environment shall be clearly specified.

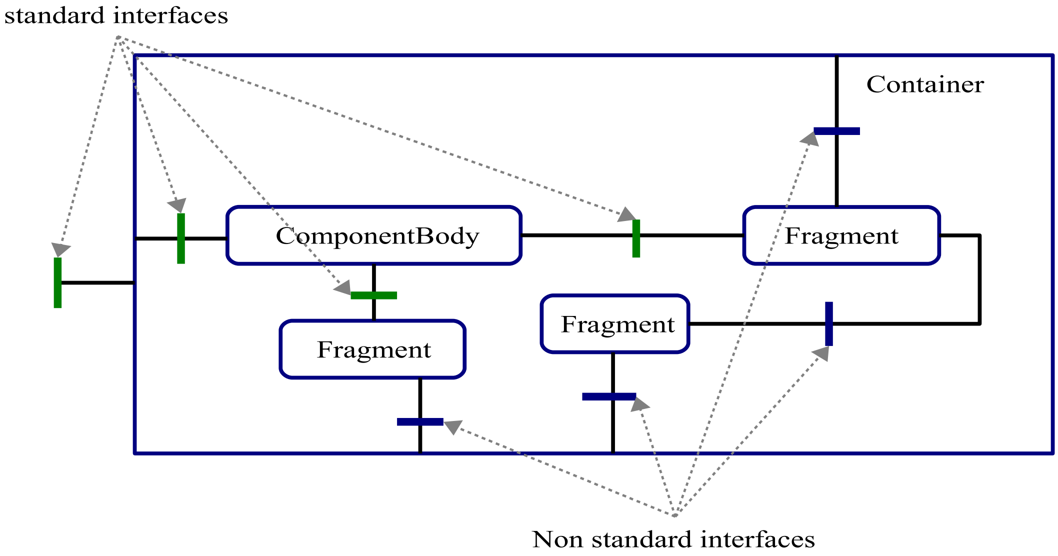

Figure 36: UCM Runtime Interfaces

Figure 37 depicts the UML model of the UCM container programming model elements. They are described in the following.

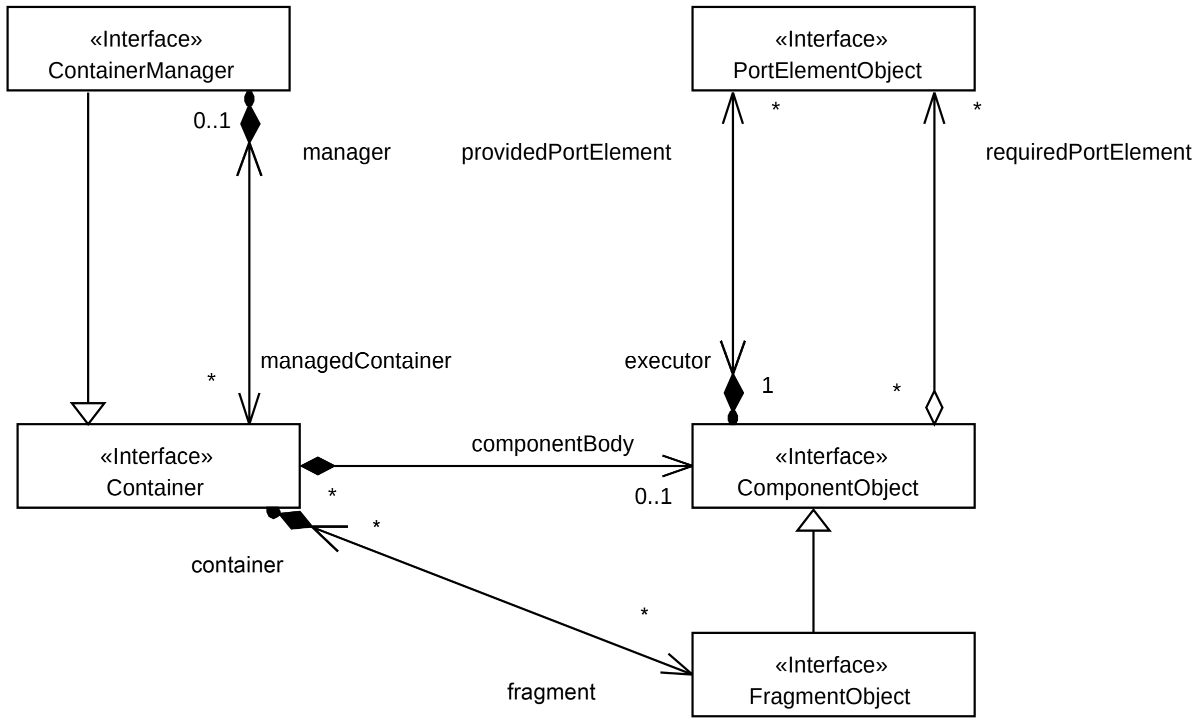

Figure 37: UCM container model

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

14.2.1 Component interfaces

Figure 38 shows the different interfaces of a UCM component whatever it is a functional component or a fragment.

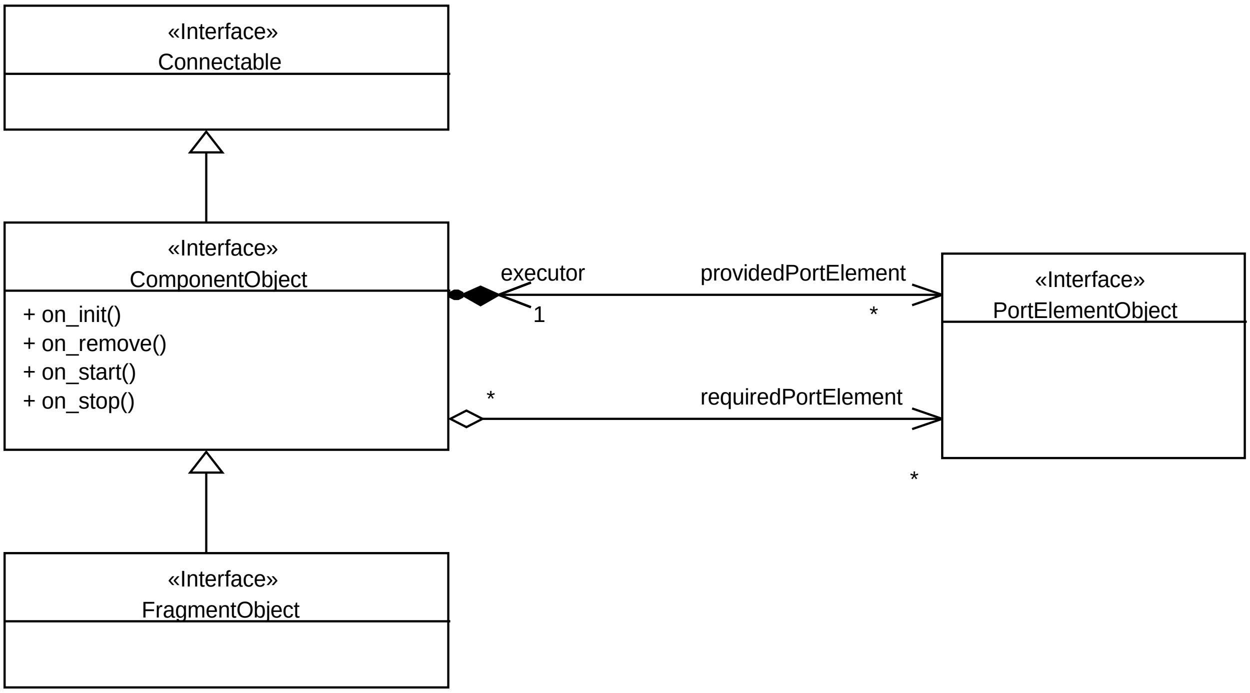

Figure 38: UCM component object interfaces

14.2.1.1 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. It holds the business logic of that interface.

14.2.1.2 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 life cycle changes from its creation to its removal (§ 14.2.3) 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_start()** 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.

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

14.2.1.3 Connectable

The *Connectable* interface is a callback interface that the component body shall implement if it has required port elements in order 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 shall 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.

The exact signatures and names of methods on_connect() and on_disconnect() shall depend on the programming language mapping. A language mapping may choose to define several methods to connect and disconnect, with various signature and various names. The only requirement is that language mappings shall define methods that are called upon connection and disconnection of *PortElementObject*.

14.2.1.4 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 a *FragmentObject* and a *ComponentObject* is a *FragmentObject* has the ability to access its container interface in order to collaborate with it when needed.

The FragmentObject interface shall define a method to set the container. This method is called by the container to provide the fragment of an access point to itself, so that it may get information about the belonging *ComponentObjects* and act on them if needed. The exact name and signature of this method shall depend on the programming language mapping.

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 should exhibit to its fragments is what it already exhibits to the deployment tool (the *Container* interface).

14.2.2 Container interfaces

Figure 39 depicts the Container-related interfaces.

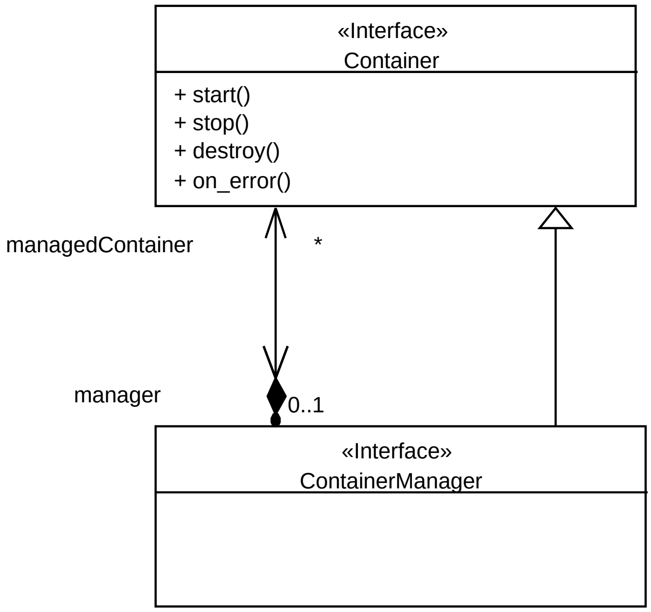

Figure 39: Container and container manager

14.2.2.1 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 *ComponentObject* and *FragmentObject* instances and manage their lifecycle. It provides a set of methods that allows to instantiate, initialize, connect and start these entities. They are described in the following items.

The following methods are mandatory and correspond to the container life cycle. Their names and signatures shall not be changed:

Method **start()** signals the completion of the configuration phase and the beginning of the operational phase for all the *ComponentObjects* of the container. Method **start()** shall therefore call methods on_start() of the component body and all the fragments.

Method **stop()** signals the end of the operational phase for all the *ComponentObjects* of the container. Method stop() shall therefore call methods on_stop() of the component body and all the fragments.

Method **on_error()** is meant to be invoked from *FragmentObjects* or subcontainers whenever they encounter an error that prevents their correct initialization or execution.

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

The following methods are mandatory, but their exact names and signatures shall depend on the language mappings.

Method **connect()** allows to associate a *ComponentObject with a PortElementObject* provided by another *ComponentObject*.

Method **disconnect()** allows to cancel a *PortElementObject* association previously established by **connect()**.

Additional methods may be defined to add or remove *ComponentObjects*, retrieve *ComponentObjects* or *PortElementObjects*, etc. The names and signatures of these additional methods shall depend on the programming language mappings.

14.2.2.2 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 may providemethods to create and remove sub-containers. The exact names and signatures of these methods shall depend on the programming language mappings. The removal of a container implies stopping and removing its contained entities (sub-containers, component bodies and fragments).

Methods **start()**, **stop()** and **destroy()** impact sub-containers: method **start()** of a container manager calls the **start()** methods of the sub-containers. Same thing for **stop()** and **destroy()**.

14.2.3 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 shall not 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 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 may 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 40 Shows the different states that the component passes through during these two phases.

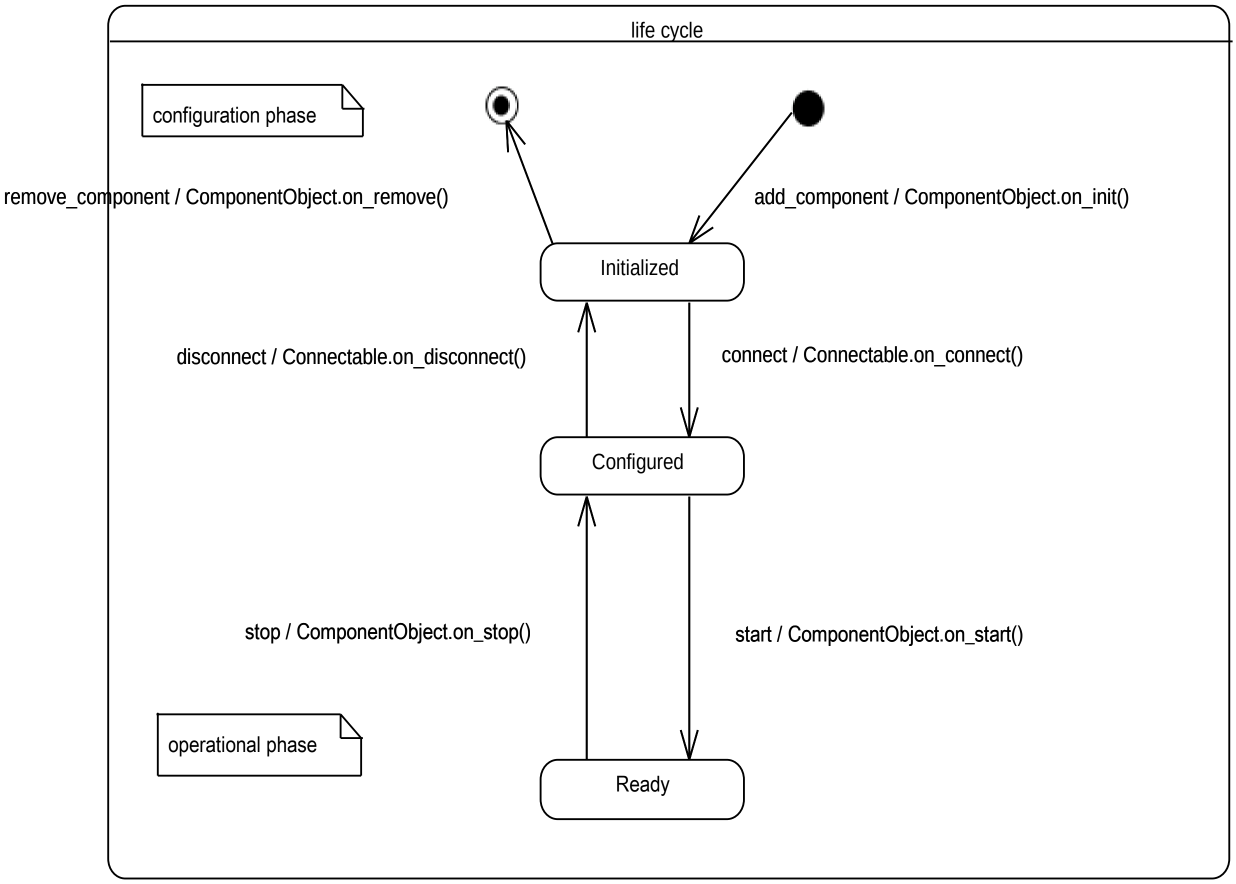

Figure 40: UCM component instance life cycle

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

The component instance is *initialized* when the container adds the component body. The component body is then instantiated and its attributes are set. To finalize the component initialization, the container shall call the **on_init()** method on the component body. The container shall also add all the fragments and call **on_init()** on them.

Once the container has added and initialized all its *ComponentObjects* (the component body and the fragments), the component instance is *configured* after successive calls to the **connect()** container methods 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()** methods. The component instance may come back to the ***initialized*** state if all its connections are undone upon calls to the **disconnect()** methods 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 be stopped 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 and the fragments of their stop.

A component instance may be removed at any time using a call to the **remove()** method of the container. To be removed, an instance shall be stopped then disconnected first.

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

This page intentionally left blank.

15. 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.

15.1 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

15.2 General notes on data types mapping

The mapping of data types relies on existing IDL types augmented with UCM-dedicated annotations where needed. 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 15.4 for more details.

15.3 Primitive types mapping

15.3.1 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
OCTET	octet
INT8	int8 (in IDL 4.2)
INT16	short
INT32	long
INT64	long long
UINT8	uint8 (in IDL 4.2)
UINT16	unsigned short
UINT32	unsigned long

UINT64	unsigned long
FLOAT	float
DOUBLE	double
CHAR8	char
WCHAR	wchar
BOOLEAN	bool

15.4 Complex data types mapping

15.4.1 Mapping to IDL constructed types

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

15.4.1.1 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 memory_footprint {
   unsigned long max;
}
```

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

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

15.4.1.2 Annotation for specifying default values

Use the standard IDL annotation @value.

15.5 Constants mapping

A UCM constant is simply translated to an IDL constant.

15.6 Interfaces and exceptions mapping

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

UCM interfaces are 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.

15.7 UCM module 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*, *NonfunctionalAspectModule*, 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 an abstract type definition (IAbstractTypeDeclaration-derived meta-classes), the corresponding IDL module becomes a template module whose parameter is the IModule abstract type. The IDL template parameter name shall be the capitalized name of the IModule abstract type. If this parameter is an *AbstractInterface*, the idl parameter shall be tagged as an "*interface*". If it is *an AbstractDataType*, the idl module parameter shall be tagged generically with the "*typename*" keyword. If a UCM *IModule* includes an element that uses an AbstractTypeDeclaration defined in a different UCM *IModule*, its equivalent module becomes a template one as well.

Example

In UCM:

```
<interactionModule name="services">
  <contractModule name="contracts">
    <AbstractInterface name="service_intf_t"/>
  </contractModule>

  <contractModule name="data">
    <abstractDataType name="message_type_t"/>
  </contractModule>

  <portType name="service_server_port">
    <portElement name="api" interface="contracts::service_intf_t" kind="provided"/>
  </portType>
</interactionModule>
```

In IDL:

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

 module data<typename MESSAGE_TYPE_T>
 {
 };

...

};
```

15.8 Component Mapping

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

15.8.1 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 15.8.4

Example

UCM:

```
<compType name="TellerComponent">
  <attribute name="id" type="::data::short_t" kind="read"/>
  <port name="teller">
  …
  </port>
</compType>
```

IDL mapping:

```
interface TellerComponent {
  readonly attribute ::data::short_t id;
  // ports mapping operations
};
```

15.8.2 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 (§ 14.2.1.2).

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:

```
<atomic language="::lang::cpp" name="TellerComponentImpl" type="TellerComponent">
…
</atomic>
<atomic language="::lang::cpp" name="PrinterComponentImpl" type="PrinterComponent">
…
</atomic>
```

IDL:

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

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

15.8.3 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 (§14.2.1.1).

15.8.4 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 is associated to *an IPortType* that has been defined as an abstract element in the UCM meta-model. The *PortType* element has been proposed as a possible way to specify a Port API. The present mapping applies on *PortType* port types only.

Each Port having a PortType 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 (§ 13.1.6) 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
};
```

15.8.5 Technical Policy Mapping

Technical policies may specify provided or required interfaces. Provided interfaces shall be implemented by the component code and the required ones shall 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.

```
<policy name="ExecPolicy" def="::policy_mod::self_exec_comp">
…
</policy>

<policy name="TracePolicy" def="::policy_mod::component_trace">
…
</policy>

<atomic language="cpp" name="TellerComponentImpl" type="TellerComponent">
  <policy name=”ExecPolicy”/>
  <policy name=”TracePolicy”/>
</atomic>

<atomic language="::lang::cpp" name="PrinterComponentImpl" type="PrinterComponent">
  <policy name=”ExecPolicy” />
  <policy name=”TracePolicy” />
</atomic>
```

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
};
```

15.9 Interaction Definition 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 ***PortType*** elements.

15.10 Container model

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

```
// base file for the UCM container model

module ucm
{
 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
 {
 };


 interface ComponentObject
 {
  void on_init ();

  void on_remove ();

  void on_start ();

  void on_stop ();

  PortElementObject get_portElement (in string provided);
 };


 interface Container;
```

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


  interface Connectable
  {
    void on_connect (in string port_or_policy, in string port_elem, in PortElementObject required);

    void on_disconnect (in string port_or_policy, in string port_elem);
  };


  interface Container
  {
    componentId add_component (in string name, 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_or_policy, in
string port_elem)
      raises (NOT_FOUND, UCM_ERROR);

    connectionId connect (in componentId instance, in string port_or_policy, in string port_elem, 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);

    void destroy ()
      raises (UCM_ERROR);

    void on_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);

  };

};          // end of module ucm
```

15.11 Component implementation 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 41 depicts this requirement.

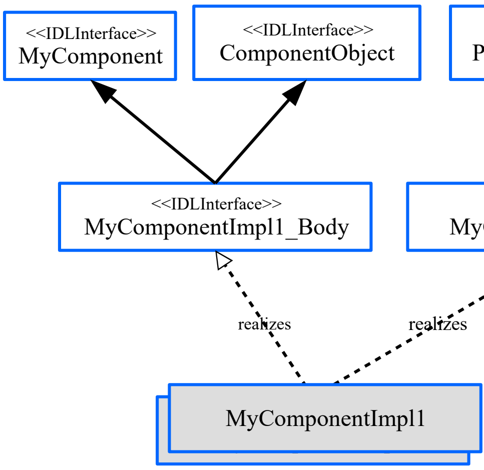

Figure 41: generated IDL interfaces

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

The component body implementation shall 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 shall implement the business methods of the related interface.

All the UCM interfaces shall 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.

15.11.1 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.

This page intentionally left blank.

16. 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.

16.1 Primitive types mapping

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

UCM primitive type	C++ primitive type
OCTET	uint8_t
INT8	int8_t
INT16	int16_t
INT32	int32_t
INT64	int64_t
UINT8	uint8_t
UINT16	uint16_t
UINT32	uint32_t
UINT64	uint64_t
FLOAT	float
DOUBLE	double
CHAR8	char
WCHAR	wchar_t
BOOLEAN	bool

16.2 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
StringType based on CHAR8	std::string
StringType based on WCHAR	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.

16.2.1 Structure mapping

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

16.2.2 Union mapping

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

16.2.3 Enumeration mapping

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

An example is given below. In UCM:

```
<enum name=”Shape” indexType=”ushort”>
  <value name=”triangle” index=”10”/>
  <value name=”square” index=”20”/>
  <value name=”circle” index=”30”/>
</enum>
```

In C++:

```
enum Shape : uint16_t {
  triangle = 10,
```

```
  square = 20,
  circle = 30
  };
};
```

16.2.4 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++.

16.2.5 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++.

16.2.6 String mapping

A UCM CHAR8 base string type maps to std::string. A UCM WCHAR base string type maps to std::wstring.

16.2.7 Constant mapping

A UCM constant maps to a C++ constant.

16.3 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.

16.4 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.

16.5 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_t" mode="read"/>
 <attribute name="level" type="LogLevelsEnum"/>
</iInterface>
```

In C++:

```
class Logger {
public:
  virtual string8_t get_name() = 0;
  virtual get_LogLevelsEnum level() = 0;
  virtual void set_level(LogLevelsEnum l) = 0;
};
```

16.6 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 extends other interfaces, its equivalent C++ class will also inherit from the equivalent C++ classes of the base interfaces, using a public inheritance.

16.6.1 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

16.6.2 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.

16.7 Component,fragment and port element mapping

A component mapping defines the set of public methods that can be invoked by the container to interact with ComponentObjects, FragmentObjects and PortElementObjects. The standards define two possible component mappings: a generic mapping and a strongly typed mapping. Tool may alternative, non-standard mappings.

Both the UCM AtomicComponentImplementation and its related ComponentType map on a common C++ class. This class represents the interface that should be implemented by the component body. It includes a set of methods corresponding to the component attributes, ports and technical policies. It also defines methods for the component object life cycle and the port connection notifications.

The class name is the same as the related atomic component implementation, suffixed with "_body". and defines four public methods to manage the component object life cycle: It extends class ComponentObject, which may be empty.

- void on_start()
- void on_init()
- void on_stop()
- void on_remove()

For each attribute declared in the component type, the class shall have two public methods: <attribute_type> get_<attribute_name>() and void set_<attribute_name>(<attribute_type> value).

Fragments inherit from class FragmentObject, which itself extends ComponentObject. They are similar to the component body class, except their constructor shall take one parameter to set a reference to the container.

Each port element is translated into a class that inherit from class PortElementObject. The class also extends the C++ class that is translated from the port element interface. The name of the port element class is <component_implementation_name>_<port_or_policy_name>_<port_element_name> for port elements provided by the component, and <fragment_name>_<port_element_name> for port elements provided by fragments.

16.7.1 Generic mapping

The generic C++11 mapping corresponds to programming language ::ucm_lang::cpp::cpp11_generic declared in the standard UCM library.

The class shall have two generic methods PortElementObjectPtr get_provided_port_element(std::string port_or_policy, std::string port_elem) and PortElementObjectPtr get_required_port_element(std::string port_or_policy, std::string port_elem).

The generic mapping also includes the port element connection methods to handle required port element connection:

- void on_connect (std::string port_or_policy, std::string port_elem, const PortElementObjectPtr required);

- void on_disconnect (std::string port_or_policy, std::string port_elem);

In UCM:

```
<bindingSet name="msg_bd">
  <binding abstract="::ucm_core::messages::api::coordinate_t"
           actual="dt::coordinate_t"/>
</bindingSet>
<compType name="Filter">
  <port name="Filter_in" type="::ucm_core::messages::msg_rcvr_pt" bindings="msg_bd"/>

  <port name="Filter_out" type="::ucm_core::messages::msg_emtr_pt" bindings="msg_bd"/>

</compType>
<atomic language="::ucm_lang::cpp11_generic" name="Filter_Impl" type="Filter">
  <policyRef ref="clock_policy"/>
</atomic>
<policy def="::ucm_core::basic_svc::clock::clock" name="clock_policy">
  <componentRef ref="Filter_Impl"/>
</policy>
```

In C++:

```
…
class PortElementObject {};
class ComponentObject {};
class FragmentObject : ComponentObject {};
typedef std::shared_ptr<PortElementObject> PortElementObjectPtr;
typedef std::shared_ptr<Container> ContainerPtr;
typedef std::shared_ptr<ComponentObject> ComponentObjectPtr;
typedef std::shared_ptr<FragmentObject> FragmentObjectPtr;


// the port element of Filter_in_rcvr_pe
class Filter_in_rcvr_pe: public PortElementObject, msg_rcvr_intf_msg_bd
{
public:
  Filter_in_rcvr_pe (ComponentObjectPtr executor);
  virtual ~Filter_in_rcvr_pe();

  void push(coordinate_t const& message);
}

class Filter_Impl_body: public ComponentObject {

public:
Filter_Impl_body();
virtual ~Filter_Impl_body();

// life cycle methods
void on_init();
void on_start ();
void on_stop();
```

```
void on_remove();

// port element connection methods
void on_connect (std::string port_or_policy, std::string port_elem, const
PortElementObjectPtr required);
void on_disconnect (std::string port_or_policy, std::string port_elem);

// access to provided port elements (for the container)
PortElementObjectPtr get_provided_port_element(std::string port_or_policy,
std::string port_elem);

// access to required port elements (for the component port elements)
PortElementObjectPtr get_required_port_element(std::string port_or_policy,
std::string port_elem);};
…

// the port element of clock
class clock_clk : public PortElementObject, public clk_intf {
public:
clock_clk(FragmentObjectPtr executor);
virtual ~clock_clk();
void get_local_time(ucm_timeval_t &local_time);
void get_synchronized_time(ucm_timeval_t &synchronized_time);
};

class clock_body : public FragmentObject {
public:
clock_body(ContainerPtr container);
virtual ~clock_body();

// life cycle methods
void on_init();
void on_start();
void on_stop();
void on_remove();

// access to provided port elements (for the container)
PortElementObjectPtr get_required_port_element(std::string port_or_policy,
std::string port_elem);
};
```

16.7.2 Strongly typed mapping

The strongly typed C++11 mapping corresponds to programming language ::ucm_lang::cpp::cpp11_typed declared in the standard UCM library.

For each port of the component type, and for each provided port element of the associated port type, the class shall have a public method <port_element_type> *get_<port_name>_<port_element_name>().

For each policy of the atomic component implementation, and for each provided port element declared in the policy definition, the class shall have a public method <port_element_type> *get_<policy_name>_<port_element_name>().

For each port of the component type, and for each required port element of the associated port type, the class shall have a public method <port_element_interface> *get_<port_name>_<port_element_name>().

For each policy of the atomic component implementation, and for each required port element declared in the policy definition, the class shall have a public method <port_element_interface> *get_<policy_name>_<port_element_name>().

The class shall have specific port element connection methods to handle required port element connection:

- void on_<port_or_policy_name>_<port_element_name>_connect (const <port_element_interface>* required);

- void on_<port_or_policy_name>_<port_element_name>_disconnect ();

In UCM:

```
<bindingSet name="msg_bd">
  <binding abstract="::ucm_core::messages::api::coordinate_t"
           actual="dt::coordinate_t"/>
</bindingSet>
<compType name="Filter">
  <port name="Filter_in" type="::ucm_core::messages::msg_rcvr_pt" bindings="msg_bd"/>

  <port name="Filter_out" type="::ucm_core::messages::msg_emtr_pt" bindings="msg_bd"/>

</compType>
<atomic language="::ucm_lang::cpp11_typed" name="Filter_Impl" type="Filter">
  <policyRef ref="clock_policy"/>
</atomic>
<policy def="::ucm_core::basic_svc::clock::clock" name="clock_policy">
  <componentRef ref="Filter_Impl"/>
</policy>
```

In C++:

```
…
class PortElementObject {};
class ComponentObject {};
class FragmentObject : public ComponentObject {};
class Container {};
typedef std::shared_ptr<PortElementObject> PortElementObjectPtr;
typedef std::shared_ptr<Container> ContainerPtr;
typedef std::shared_ptr<ComponentObject> ComponentObjectPtr;
typedef std::shared_ptr<FragmentObject> FragmentObjectPtr;

class Filter_Impl_body;

// the port element of Filter_in_rcvr_pe
class Filter_in_rcvr_pe: public PortElementObject, public msg_rcvr_intf_msg_bd
{
public:
  Filter_in_rcvr_pe (Filter_Impl_body *executor);
  virtual ~Filter_in_rcvr_pe();

  void push(coordinate_t const& message);
};

class Filter_Impl_body: public ComponentObject
{
public:
Filter_Impl_body();
virtual ~Filter_Impl_body();
```

```
// life cycle methods
void on_init();
void on_start ();
void on_stop();
void on_remove();

// port element connection methods
void on_Filter_out_emtr_pe_connect (const msg_emtr_intf_msg_bd* required);
void on_Filter_out_emtr_pe_disconnect ();
void on_clock_policy_clk_connect (const clk_intf *required);
void on_clock_policy_clk_disconnect ();

// access to provided port elements (for the container)
Filter_in_rcvr_pe *get_Filter_in_rcvr_pe();

// access to required port elements (for the component port elements)
msg_emtr_intf *get_Filter_out_emtr_pe();
clk_intf *get_clock_policy_clk();
};
…

class clock;

// the port element of clock
class clock_clk : public PortElementObject, public clk_intf {
public:
clock_clk(clock *executor);
virtual ~clock_clk();
void get_local_time(ucm_timeval_t &local_time);
void get_synchronized_time(ucm_timeval_t &synchronized_time);
};

class clock_body : public FragmentObject {
public:
clock_body(ContainerPtr container);
virtual ~clock_body();

// life cycle methods
void on_init();
void on_start();
void on_stop();
void on_remove();

// access to provided port elements (for the container)
PortElementObjectPtr get_required_port_element(std::string port_or_policy,
std::string port_elem);
};
```

16.8 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 shall 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.

16.8.1 Array mapping

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

16.8.2 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:

```
<enum name=”Shape” indexType=”uint16”>
  <value name=”triangle” index=”10”/>
  <value name=”square” index=”20”/>
  <value name=”circle” index=”30”/>
</enum>
```

In C++:

```
struct Shape_def {
 enum type {
    triangle = 10,
    square = 20,
    circle = 30
  };
};
typedef ucm::safe_enum<Shape_def, uin16_t>
        Shape;

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

16.8.3 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.

This page intentionally left blank.

17. XML examples of UCM declarations (non-normative)

This section is illustrative. It provides examples of the XML syntax for UCM declarations. All cases are not covered; the purpose of this section is only to show typical example to help understand the syntax.

17.1 Contracts

17.1.1 Standard data types: primitive data types

All UCM primitive have a name. Float, integer and char types also have a kind; the allowed values for the type kinds are the ones defined in section 9.2.3.

```
<contractModule name="primitive_types">
  <integer kind="int16" name="an_interger_type"/>
  <bool name="a_boolean_type"/>
  <octet name="an_octet_type"/>
  <float kind="double" name="a_float_type"/>
  <char kind="wchar" name="a_character_type"/>
</contractModule>
```

17.1.2 Standard data types: complex types

UCM complex types are enumerations, structures, unions and aliases. Enumerations have an integer index. Unions are discriminated by an enumeration.

```
<contractModule name="complex_types">
  <enum indexType="uint32" name="coord_t">
    <value index="0" name="cartesian"/>
    <value index="1" name="polar"/>
  </enum>
  <struct name="cartesian_t">
    <field name="x" type="::primitive_types::an_interger_type"/>
    <field name="y" type="::primitive_types::an_interger_type"/>
  </struct>
  <union name="coordinate" selector="system" selectorType="coord_t">
    <case default="true" name="cart" type="cartesian_t" when="cartesian"/>
    <case name="pol" type="polar_t" when="polar"/>
  </union>
  <struct name="polar_t">
    <field name="theta" type="::primitive_types::a_float_type"/>
    <field name="r" type="::primitive_types::a_float_type"/>
  </struct>
  <alias name="rectangular_t" type="cartesian_t"/>
  <array name="image_t" type="color_16bits">
    <comment>800x600-16bit image</comment>
    <dim indexType="uint32" size="480000"/>
  </array>
  <integer kind="uint16" name="color_16bits"/>
</contractModule>
```

17.1.3 Standard data types: resizable types

Resizable types are strings, sequences and native types. Strings specify the base character kind. Sequences are indexed by an integer kind.

```
<contractModule name="resizable_types">
  <string base="char8" maxSize="12" name="message_t"/>
  <sequence indexType="uint32" maxSize="7" name="message_seq" type="message_t"/>
```

```
  <native maxSize="128" name="native_type"/>
</contractModule>
```

17.1.4 Constants

```
Constants have a name, a type and a value.
<contractModule name="constants">
  <constant name="pi" type="::primitive_types::a_float_type" value="3.14"/>
</contractModule>
```

17.1.5 Interface, methods and exceptions

Interfaces may extend other interfaces.

```
<contractModule name="interfaces">
  <integer kind="int16" name="error_code_t"/>
  <sequence indexType="uint32" maxSize="256" name="buffer_t"
type="::primitive_types::an_octet_type"/>
  <exception name="runtime_error">
    <excepField name="error_code" type="error_code_t"/>
  </exception>
  <interface name="read_intf">
    <method name="read">
      <excep ref="runtime_error"/>
      <param dir="inout" name="buffer" type="buffer_t"/>
      <param dir="out" name="size" type="::primitive_types::an_interger_type"/>
    </method>
  </interface>
  <interface name="read_write_intf">
    <extends ref="read_intf"/>
    <method name="write_one">
      <excep ref="runtime_error"/>
      <param dir="in" name="data" type="::primitive_types::an_octet_type"/>
    </method>
    <method name="read_one">
      <param dir="return" name="returns" type="::primitive_types::an_octet_type"/>
    </method>
  </interface>
</contractModule>
```

17.1.6 Abstract type declarations

Abstract data types and interfaces are declared like normal types. Complex and resizable types use abstract types like any other type.

```
<contractModule name="abstract">
  <abstractDataType name="abstract_type"/>
  <struct name="structure_template">
    <field name="id" type="id_t"/>
    <field name="payload" type="abstract_type"/>
  </struct>
  <integer kind="int64" name="id_t"/>
  <abstractInterface name="abstract_interface"/>
</contractModule>
```

17.1.7 Annotation and configuration elements

Annotation definitions contain configuration parameters.

```
<contractModule name="config">
  <integer kind="int16" name="fifo_size_t"/>
  <annotationDef name="expected_period">
    <configParam defaultValue="{25, ms}" name="period_param"
type="::ucm_ext_exec::contracts::ucm_duration_t"/>
  </annotationDef>
</contractModule>
```

17.2 Interactions

Section 13 contains the definitions of the standard technical aspects and policies.

17.2.1 Connector extension

A connector definition may extend an existing one. This allows for the addition of configuration parameters for a specific connector implementation.

```
<interactionModule name="connectors">
  <contractModule name="conf">
    <integer kind="int32" name="fifo_size_t"/>
  </contractModule>
  <connectorDef name="fifo_msg_cnt" pattern="::ucm_core::messages::msg_intr_pat">
    <extends ref="::ucm_core::messages::simple_msg_cnt"/>
    <portConf port="emitter">
      <configParam name="output_fifo_size" type="conf::fifo_size_t"/>
    </portConf>
  </connectorDef>
</interactionModule>
```

17.3 Nonfunctional aspects

17.3.1 Technical aspects and technical policies

Section 13 contains the definitions of the standard technical aspects and policies.

17.3.2 Supported programming languages

Supported programming language are mere names. Such names may be anything and carry no meaning in themselves. UCM tool chains should ship with a set of supported language names and be able to handle these names.

```
<policyModule name="lang">
  <language name="Cpp11"/>
</policyModule>
```

17.4 Components

The examples of this section correspond to the example illustrated in section Error: Reference source not found

17.4.1 Component types

Component type Detector defines a component type meant to detect the position of an object in an image. It has two ports. Port “detector_in” is associated with the standard message reception port type and bound to the image_t type declared in section 17.1.2. Port “detector_out” in associated with the standard message emission port type, and bound to type cartesian_t.

This component is therefore meant to take either polar or cartesian coordinates as input, and produce only cartesian coordinates.

```
<componentModule name="comp_types">
  <compType name="Detector">
    <comment>Finds the position of an object in an image</comment>
    <port bindings="image_msg" name="detector_in"
type="::ucm_core::messages::msg_rcvr_pt"/>
    <port bindings="coord_msg" name="detector_out"
type="::ucm_core::messages::msg_emtr_pt"/>
  </compType>
  <bindingSet name="image_msg">
    <binding abstract="::ucm_core::messages::api::transm_data_t"
actual="::complex_types::image_t"/>
  </bindingSet>
  <bindingSet name="coord_msg">
    <binding abstract="::ucm_core::messages::api::transm_data_t"
actual="::complex_types::cartesian_t"/>
  </bindingSet>
</componentModule>
```

17.4.2 Atomic component implementations and technical policies

Atomic component “Detector_atomic” is an implementation of component type “Detector”. It is associated with technical policy “exec_policy”; this association is named “exec”, like the input port of Filter is named “in”. Technical policy “exec_policy” also has a link to “Detector_atomic”.

The component implementation also contains an annotation to specify an expected execution period. This may be used to complement the functional contracts defined in the component type (i.e. the port type specifications) by specifying additional nonfunctional contracts.

```
<componentModule name="atomic">
  <atomic lang="::lang::Cpp11" name="Detector_atomic" type="::comp_types::Detector">
    <policyRef ref="exec_policy"/>
    <annotation def="::config::expected_period">
      <config def="period_param" value="{27, ms}"/>
    </annotation>
  </atomic>
  <policy def="::ucm_core::comp_exec::prot_actv_comp" name="exec_policy">
    <componentRef ref="Detector_atomic"/>
  </policy>
</componentModule>
```

17.4.3 Composite component implementations

The following declarations define a composite filter made of two subcomponents: the detector itself, and a filter which performs some preprocessing on the image.

Composite component “Detector_composite” contains two subcomponents, named “filter” and “detector”. Connection “cnt1” connects port “filter_out” of “filter” and port “detector_in” of “detector”. Port “detector_in” of “Detector_composite” is delegated to port “filter_in” of “filter”; port “detector_out” of “Detector_composite” is delegated to port “detector_out” of “detector”.

```
<componentModule name="composite">
  <composite name="Detector_composite" type="::comp_types::Detector">
    <part name="detector" ref="::atomic::Detector_atomic"/>
    <part name="filter" ref="Filter_atomic"/>
    <connection name="cnt1" ref="::ucm_core::messages::simple_msg_cnt">
      <end name="cnt1_filter" part="filter" port="filter_out"/>
      <end name="cnt1_detector" part="detector" port="detector_in"/>
    </connection>
    <portDelegation extPort="detector_in" name="filter_in" part="filter"
port="filter_in"/>
    <portDelegation extPort="detector_out" name="filter_out" part="detector"
port="detector_out"/>
  </composite>
  <atomic lang="::lang::Cpp11" name="Filter_atomic" type="Filter">
    <policyRef ref="passive_filter"/>
  </atomic>
  <policy def="::ucm_core::comp_exec::unpr_pasv_comp" name="passive_filter">
    <componentRef ref="Filter_atomic"/>
  </policy>
  <compType name="Filter">
    <port bindings="::comp_types::image_msg" name="filter_in"
type="::ucm_core::messages::msg_rcvr_pt"/>
    <port bindings="::comp_types::image_msg" name="filter_out"
type="::ucm_core::messages::msg_emtr_pt"/>
  </compType>
</componentModule>
```