Date: April 2008

Robotic Technology Component Specification

Version 1.0

OMG Document Number: formal/2008-04-04
Standard document URL: http://www.omg.org/spec/RTC/1.0/PDF
Associated files*: http://www.omg.org/spec/RTC/20070901 (XMI)
http://www.omg.org/spec/RTC/20070801 (IDL)
http://www.omg.org/spec/RTC/20070802 (C++ Header)

* original files: ptc/07-09-10 (XMI), ptc/07-08-21 (IDL), and ptc/07-08-22 (C++ Header)

OMG's Issue Reporting Procedure

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

Table of Contents

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG 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
140 Kendrick Street
Building A, Suite 300
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 - 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 *http://www.omg.org/technology/agreement.htm.*

1 Scope

1.1 Overview

This document defines a component model and certain important infrastructure services applicable to the domain of robotics software development. It includes a Platform-Independent Model (PIM) expressed in UML and Platform-Specific Models (PSMs) expressed in OMG IDL.

The sophistication of robotic and other intelligent systems has increased in recent years, allowing such systems to be applied to an ever-increasing range of problems. Such systems are no longer constrained to the factory floor, but are now used in a variety of commercial and non-commercial applications. They have even begun to enter our homes. The technologies used in these applications, which can be applied not only to standalone robots but also to ubiquitous computing and other more intelligent electrical devices, we call “Robotic Technology” (RT).

As the level of structural and behavioral complexity in RT systems has increased, so too has the need for technologies supporting the integration of those systems. A systematic methodology and infrastructure for the local and distributed composition of modular functional components is needed.

This document specifies a component model that meets these requirements of RT systems. We refer to a component that supports the integration of RT systems as a Robotic Technology Component (RTC) [RTC RFP]. An RTC is a logical representation of a hardware and/or software entity that provides well-known functionality and services. By extending the general-purpose component functionality of UML with direct support for domain-specific structural and behavioral design patterns, RTCs can serve as powerful building blocks in an RT system. Developers can combine RTCs from multiple vendors into a single application, allowing them to create more flexible designs more quickly than before.

This specification does not seek to replace existing component models (e.g., UML components, Lightweight CCM, Software Radio components). Instead, it focuses on those structural and behavioral features required by RT applications that are not addressed by pre- existing models. It is anticipated that some implementers of this specification will choose to implement it in terms of another, more general, component model.

1.2 Platform-Independent Model

The PIM described herein is applicable to a very wide range of robotics and controls applications, from fine-grained local components to coarser-grained components communicating over a network. Its focus is at a somewhat higher level of abstraction and domain-specificity than that of more general purpose models (e.g., [CCM]) and it includes direct support for several important design patterns that are central to the architecture of many robotics applications.

The PIM consists of three parts:

- Lightweight RTC. A simple model containing definitions of concepts such as component, port, and the like. This section provides the foundation on which the subsequent sections build. See Section 5.2.
- Execution semantics. Extensions to Lightweight RTC to directly support critical design patterns used in robotics applications such as periodic sampled data processing, discrete event/stimulus response processing, modes of operation, etc. See Section 5.3.
- Introspection. An API allowing for the examination of components, ports, connections, etc. at runtime. See Section 5.4.

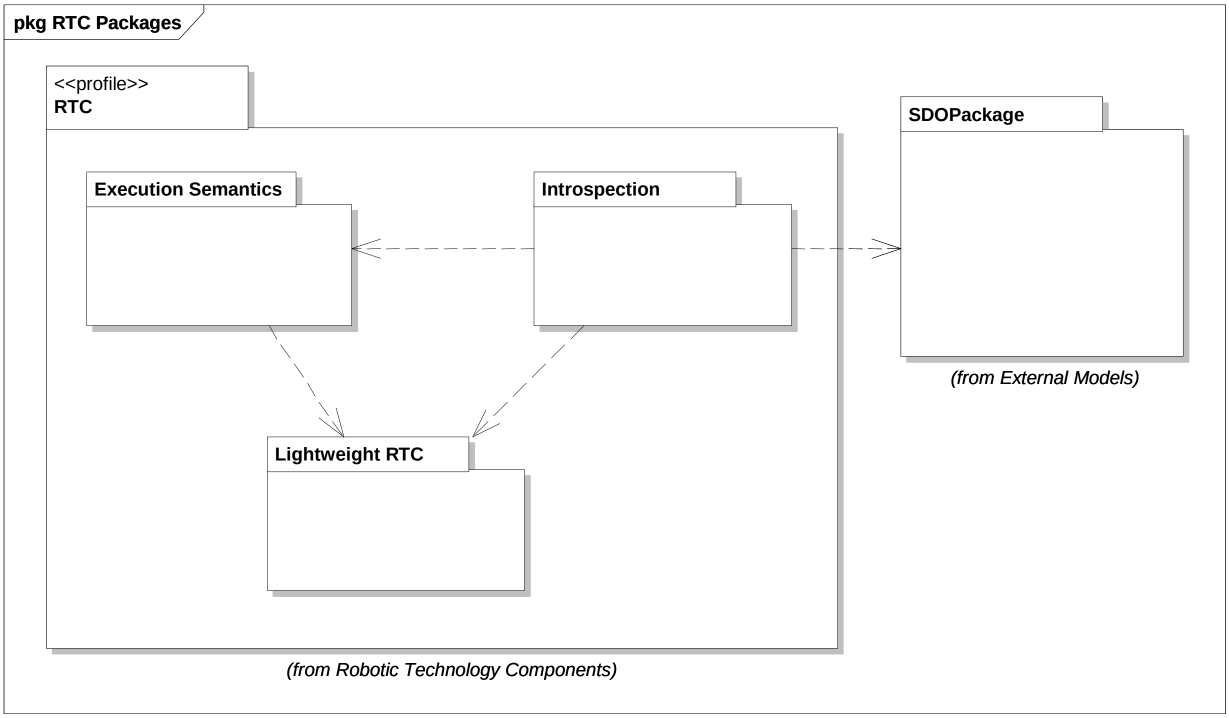

Figure 1.1 - RTC packages

Figure 1.1 shows the relationships among the sections described above.

1.3 Platform-Specific Models

Three PSMs are described:

1. Local. Components reside on the same network node and communicate over direct object references without the mediation of a network or network-centric middleware such as CORBA.
2. Lightweight CCM [CCM]. Most components are assumed to be distributed relative to one another; they communicate using a CCM-based middleware.
3. CORBA [CORBA]. Most components are assumed to be distributed relative to one another; they communicate using a CORBA-based middleware.

The Local PSM is primarily applicable to relatively fine-grained components executing within a single application. It offers the potential for very low latency, high determinism communications (on the order of a small number of function calls) and is thus appropriate for components or applications with hard real-time requirements.

The [CORBA]- and [CCM]-based PSMs are primarily applicable when most components are distributed over a network. These components are likely to be coarser grained, more autonomous, and have higher tolerances for latency, jitter, and communications failure than components that communicate locally. Example applications include groups of service robots communicating within a home and mobile phones exchanging voice calls or other data within a cellular network.

2 Conformance and Compliance

Support for Lightweight RTC (see Section 5.2) is fundamental and obligatory for all implementations.

In addition, implementations are strongly urged to support at least one of the following optional conformance points, as they provide greater domain-specific value to robotics applications than Lightweight RTC alone.

- Periodic Sampled Data Processing (see Section 5.3.1)
- Stimulus Response Processing (see Section 5.3.2)
- Modes (see Section 5.3.3)
- Introspection (see Section 5.4)

At least one of the following PSMs must be implemented for each of the conformance points in the list above to which conformance is claimed.

- Local communications (see Section 6.3)
- [CCM]-based communications (see Section 6.4)
- [CORBA]-based communications (see Section 6.5)

Interoperability across programming languages is beyond the scope of the Local PSM. Conformance is therefore relative to a particular IDL-to-programming language mapping. Providers must state the language mapping(s) for which their implementation is conformant. The following is an example statement of conformance:

> *Middleware X* is conformant to the Lightweight RTC, Periodic Sampled Data Processing, and Introspection conformance points of the Robotic Technology Component (RTC) Specification on the Local platform in the C++ language, as defined by the OMG-standard IDL-to-C++ mapping, version 1.1 (formal/03-06-03).

3 References

3.1 Normative References

[CORBA] Common Object Request Broker Architecture (CORBA) Core Specification, version 3.0.3
Specification: formal/2004-03-12
http://www.omg.org/technology/documents/formal/corba_iiop.htm

[CCM] CORBA Component Model, version 4.0
http://www.omg.org/technology/documents/formal/components.htm

[RTC RFP] RFP for Robot Technology Components
http://www.omg.org/cgi-bin/doc?ptc/2005-09-01

[SDO] Super Distributed Objects
http://www.omg.org/cgi-bin/doc?formal/2004-11-01

[UML] Unified Modeling Language, Superstructure Specification version 2.1.1
http://www.omg.org/technology/documents/formal/uml.htm

3.2 Non-normative References

[CCM STREAM] Streams for CCM Specification
This specification is still in finalization as of this writing. The document number is ptc/05-07-01.
http://www.omg.org/cgi-bin/doc?ptc/2005-07-01

[CCM UML] UML Profile for CORBA Components, version 1.0
Specification: formal/05-07-06
Supporting files: formal/05-08-03
http://www.omg.org/technology/documents/formal/profile_ccm.htm

[CORBA UML] UML Profile for CORBA Specification, version 1.0
Specification: formal/2002-04-01
http://www.omg.org/technology/documents/formal/profile_corba.htm

[DC] Deployment and Configuration of Component-based Distributed Applications, version 1.0
http://www.omg.org/cgi-bin/doc?ptc/2004-08-02

[DDS] Data Distribution Service, version 1.1
http://www.omg.org/cgi-bin/doc?formal/05-12-04

[RT CORBA] RealTime-CORBA Specification, version 2.0
Specification: formal/2003-11-01
http://www.omg.org/technology/documents/formal/RT_dynamic.htm

[SWRADIO] PIM and PSM for Software Radio Components

http://www.omg.org/cgi-bin/doc?dtc/2005-09-05

4 Additional Information

4.1 Requirements

RT applications typically require the following features:

- Components of varying granularity: Software must be hierarchically decomposable to an arbitrary depth. A component model must therefore support components of varying granularity. A simple algorithm performing some mathematical or control function may be encapsulated as a fine-grained component. On the other extreme, an entire humanoid robot may be represented as a single autonomous, coarse-grained component. A component developer must be free to choose the granularity appropriate for a given component and to compose components of differing granularity within a single application.

- Autonomous and passive components: Some components must operate autonomously and in parallel with one another. Such components must own an independent task of execution in which to carry out their work (e.g., real-time feedback control). In other cases, however, separate threads for each component is an unnecessary expense (e.g., returning a value from a method invocation). The mapping of components to threads of execution must therefore be flexible.
- Real-time capabilities: Some tasks within an RT application (such as servo control) require hard real-time capabilities, including very low latencies and a high degree of determinism.
- Systematic execution: The execution ordering of modules within an application frequently follows one of a few idiomatic design patterns, such as sampled execution or finite state machines. The component model should support such design patterns.
- Software reuse: Components must be strongly encapsulated for maximum reusability.
- Network and communication independence: Different applications have different requirements with respect to inter-component communication. A component model must therefore be independent of any particular communications medium or protocol.

4.2 Relationships to Existing OMG Specifications

4.2.1 Platform-Independent Model

The PIM is defined in UML 2.1.1 [UML].

The full RTC specification extends Lightweight RTC (described in this document) and Super Distributed Objects (described in [SDO]).

4.2.2 Platform-Specific Models

The PSMs are defined in part using OMG IDL. The Lightweight CCM PSM depends on that specification [CCM].

Note that the transformation of the PIM into CCM does not rely on the UML Profile for CORBA [CORBA UML] or the UML Profile for CORBA Components [CCM UML].

- The purpose of the CCM PSM in this specification is to describe models based on UML components using CCM.
- The purpose of [CORBA UML] and [CCM UML] is the opposite: to describe CORBA- and CCM-based applications using UML . They do not use UML components or interfaces (IDL components and interfaces are represented as UML classes).

Therefore, this specification maps the relevant concepts to CCM directly. More information may be found in Chapter 6.

4.3 Acknowledgements

The following companies submitted and/or supported parts of this specification.

- Japan Robot Association (JARA)
- National Institute of Advanced Industrial Science and Technology (AIST)
- Real-Time Innovations (RTI)

- Seoul National University (SNU)
- Technologic Arts Incorporated

5 Platform Independent Model

5.1 Format and Conventions

Classes are described in this PIM using tables of the following format. A more detailed description of each member follows the table.

<class name>		
attributes		
`<attribute name>`	`<attribute type>`	
...	...	
operations		
`<operation name>`		`<return type>`
	`<parameter name>`	<parameter type>
	...	...

Enumerated constants are modeled as attributes of the enumeration type.

Operation parameters can contain a modifier "**`in`**," "**`out`**," or "**`inout`**" ahead of the parameter name. If the modifier is omitted, it is implied that the parameter is an "**`in`**" parameter.

In some cases the operation parameters and/or return value(s) are a collection with elements of a given <type>. This scenario is indicated with the notation "<type> [ ]." This notation does not imply that it will be implemented as an array. The actual implementation is defined by the PSM; it may end up being mapped to a sequence, an array, or some other kind of collection.

For example, the class named **`MyClass`** below has a single attribute named **`my_attribute`** of type **`long`** and a single operation **`my_operation`** that returns a **`long`**. The operation takes four parameters. The first, **`param1`**, is an output parameter of type **`long`**; the second, **`param2`**, an input-output parameter of type **`long`**; the third, **`param3`**, is an input parameter (the "in" modifier is implied by omission) of type **`long`**; and the fourth, **`param4`**, is also an input parameter of type collection of **`long`**s.

MyClass		
attributes		
`my_attribute`	`long`	
operations		
`my_operation`		`long`
	`out: param1`	long

	`inout: param2`	long
	`param3`	long
	`in: param4`	long[]

5.2 Lightweight RTC

The lightweight RT component model defines interfaces and stereotype to adapt UML components to the RT domain. The UML 2.0 Components and Composite Structures packages—and those packages on which they depend, directly or indirectly—are considered normative. A non-normative simplified version of that model is shown in Figure 5.1.

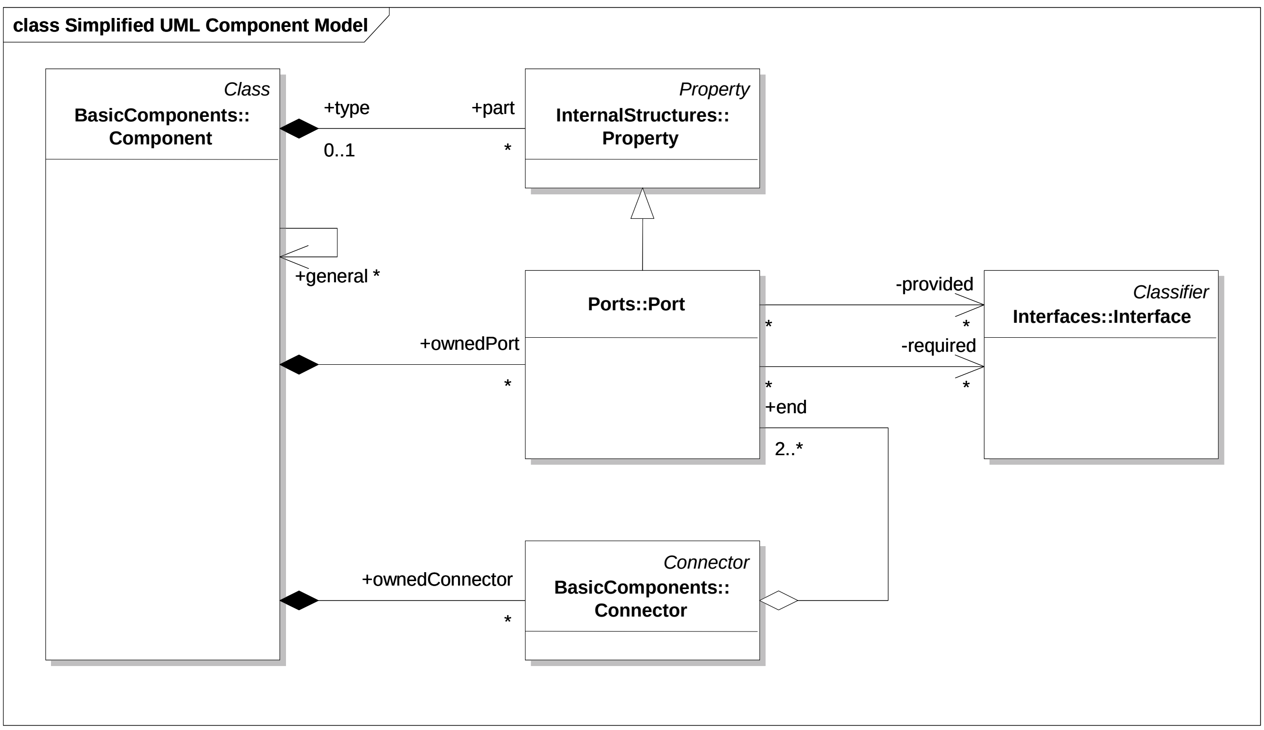

Figure 5.1 - Simplified depiction of components and their constituents from [UML]

Components

From [UML]:

> A component is a self contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces.
>
> A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality.

Ports

From [UML]:

> Ports represent interaction points between a classifier and its environment. The interfaces associated with a port specify the nature of the interactions that may occur over a port. The required interfaces of a port characterize the requests that may be made from the classifier to its environment through this port. The provided interfaces of a port characterize requests to the classifier that its environment may make through this port.

Note that a port may expose multiple interfaces, and those interfaces may be of mixed polarity (i.e., some may be required and others provided). This feature is important for the expression of callback semantics or any other two-way communication contract.

Composite Components

Component instances may be composed together to form new component types; these latter are called *composite component types* or sometimes *component assemblies*. The component instances that comprise the internal structure of a composite component are referred to as its *parts*. Those parts are connected to one another and to their containing composite component via *connectors* between their respective ports. Further detail is provided in [UML].

5.2.1 ReturnCode_t

Description

A number of operations in this specification will need to report potential error conditions to their clients. This task shall be accomplished by means of operation “return codes” of type **`ReturnCode_t`**[1].

Semantics

Operations in the PIM that do not return a value of type **`ReturnCode_t`** shall report errors in the following ways, depending on their return type:

- If an operation normally returns a positive numerical value (such as **`get_rate`**, see Section 5.2.2.6.4), it shall indicate failure by returning a negative value.
- If an operation normally returns an object reference (such as **`RTObject::get_component_profile`**, see Section 5.4.2.2.1), it shall indicate failure by returning a nil reference.

Attributes

ReturnCode_t	
attributes	
OK	ReturnCode_t
ERROR	ReturnCode_t

1. *Implementation notes*: Depending on its target programming language and execution environment, a PSM may express **`ReturnCode_t`** as an actual return value or as an exception. This specification does not mandate exceptions because many platforms that are important for robotics developers do not support them well. (This error reporting convention is also used by [DDS].)

BAD_PARAMETER	ReturnCode_t
UNSUPPORTED	ReturnCode_t
OUT_OF_RESOURCES	ReturnCode_t
PRECONDITION_NOT_MET	ReturnCode_t
no operations	

5.2.1.1 OK

Description

The operation completed successfully.

5.2.1.2 ERROR

Description

The operation failed with a generic, unspecified error.

5.2.1.3 BAD_PARAMETER

Description

The operation failed because an illegal argument was passed to it.

5.2.1.4 UNSUPPORTED

Description

The operation is unsupported by the implementation (e.g., it belongs to a compliance point that is not implemented).

5.2.1.5 OUT_OF_RESOURCES

Description

The target of the operation ran out of the resources needed to complete the operation.

5.2.1.6 PRECONDITION_NOT_MET

Description

A pre-condition for the operation was not met.

5.2.2 Components

This section defines the stereotypes and interfaces that define conforming components. In particular, any conforming component must be extended by the stereotype **`lightweightRTComponent`** or some specialization thereof (such as **`rtComponent`**; see Section 5.4.2.1). Such a component shall be referred to as a "lightweight robotic technology component," or "lightweight RTC."

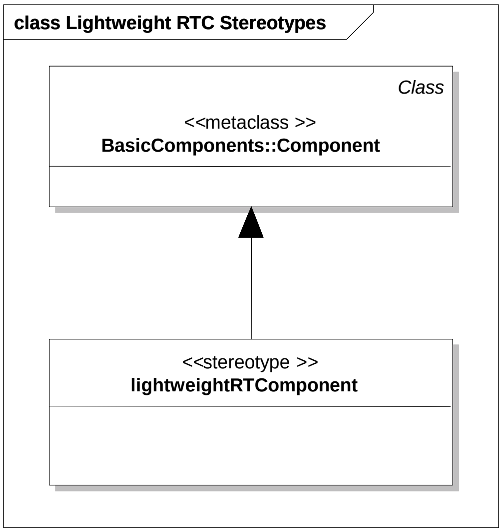

Figure 5.2 - Lightweight RTC Stereotypes

5.2.2.1 lightweightRTComponent

Description

The **`lightweightRTComponent`** stereotype extends a UML component (i.e., **`UML::Components::BasicComponents::Component`** [UML]).

A component so extended conforms to the lifecycle described in Section 5.2.2.3 and allows clients to monitor and respond to changes in that lifecycle by means of the **`ComponentAction`** interface. The following figure is a non-normative example.

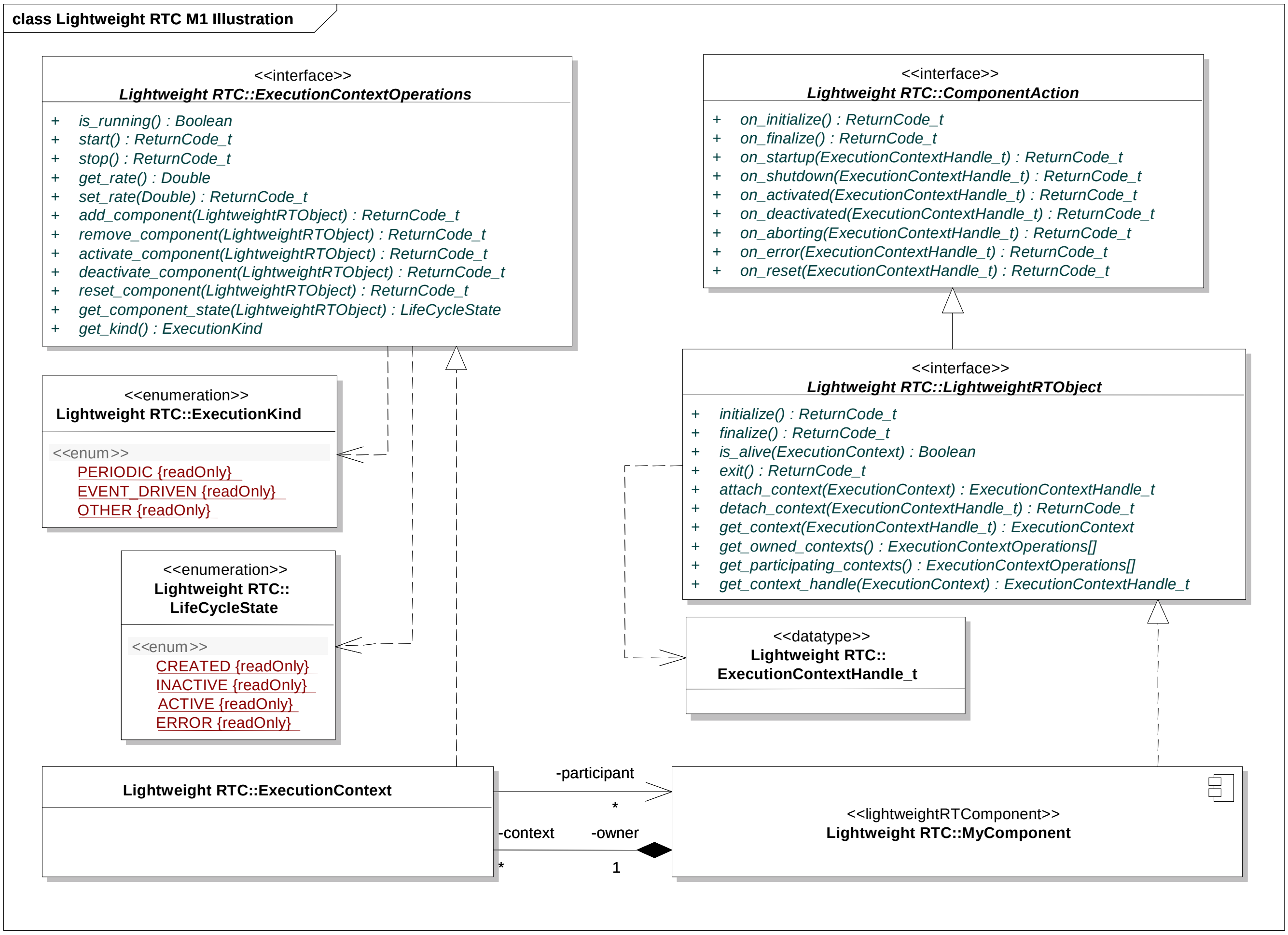

Figure 5.3 - Lightweight RTC M1 Illustration

Constraints

- A component extended by the **`lightweightRTComponent`** stereotype shall realize the **`LightweightRTObject`** interface.
- Although the UML meta-class Component allows any number of supertypes, at most one supertype may be another (lightweight) RTC.

5.2.2.2 LightweightRTObject

Description

This interface is realized by all lightweight RTCs (as required by the **`lightweightRTComponent`** stereotype). It defines the states and transitions through which all RTCs will pass from the time they are created until the time they are destroyed.

Generalizations

- **`ComponentAction`**

Semantics

Initialization

An RTC begins in the Created state; at this point, it has been instantiated but not yet fully initialized. Note that this state is highly implementation-dependent. For example, it may correspond to the invocation of a constructor in languages that support that concept, but not all languages do. Furthermore, how soon this state is entered before **`initialize`** is invoked is implementation-dependent. Therefore, it should be relied on by RTC implementers only to the minimum extent possible.

An RTC that has completed its initialization and has not been finalized is said to be Alive.

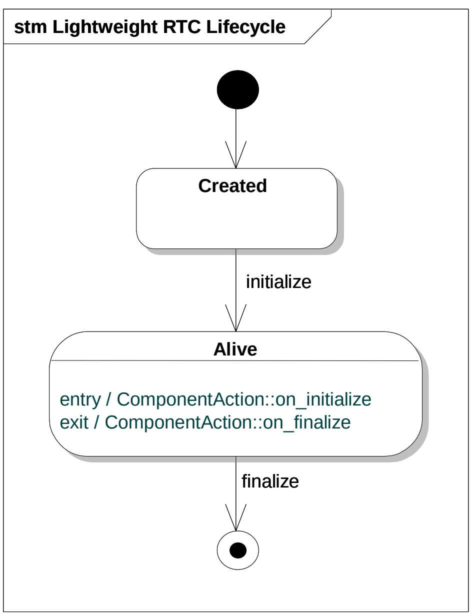

Figure 5.4 - Lightweight RTC Lifecycle

Execution Context

An RTC in the Alive state may participate in any number of execution contexts (see Section 5.2.2.5). These contexts shall be represented to an RTC as distinct instances of the **`ExecutionContext`** class. The **`ExecutionContext`** manages the behavior of each RTC that participates in it. This relationship is defined by the following state machine, which is embedded within the **`ExecutionContext`**'s own lifecycle (see Figure 5.5). Each participating RTC is represented as a separate parallel region.

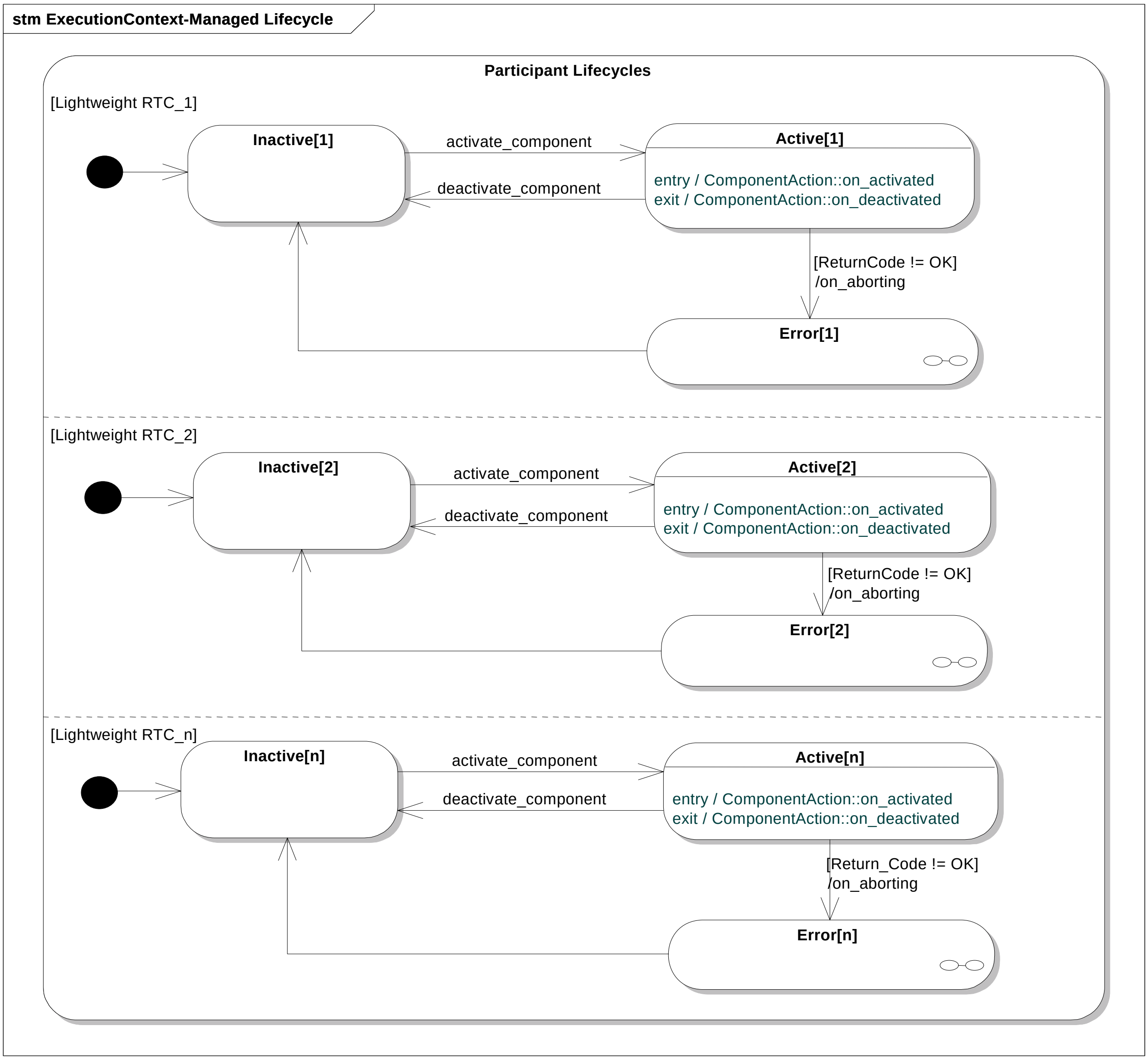

Figure 5.5 - ExecutionContext-Managed Lifecycle

Relative to a given execution context, an RTC may either be Active, Inactive, or in Error. When the RTC is Active in a Running execution context, the `ComponentAction` callbacks (see Section 5.2.2.4) shall be invoked as appropriate for the context's `ExecutionKind`. The callbacks shall not be invoked relative to that context when either the RTC is Inactive in that context or the context is Stopped. (Note that starting and stopping an execution context shall not impact whether its participating RTCs are Active or Inactive.)

It may be that a given RTC does not directly participate in any execution contexts. Such an RTC is referred to as *passive*. A passive RTC may provide services to other components upon request. At any other time, it shall not be required to perform any ongoing activity of its own; therefore, instances of such an RTC typically exist only as parts (directly or indirectly) of a containing active RTC.

Error Handling

If an operation fails while the RTC is Active in a given execution context, the RTC will transition to the Error state corresponding to that context. While the RTC is in Error, the **`ComponentAction::on_error`** callback will be invoked in place of those callbacks that would otherwise have been invoked according to the context's **`ExecutionKind`**. For example, if the kind is **`PERIODIC`**, **`on_error`** shall be invoked instead of the pair of **`on_execute`**, and **`on_state_update`**.

When an RTC is in Error, it may be reset. If resetting is successful, the RTC shall return to the Inactive state. If resetting is unsuccessful, it shall remain in the Error state.

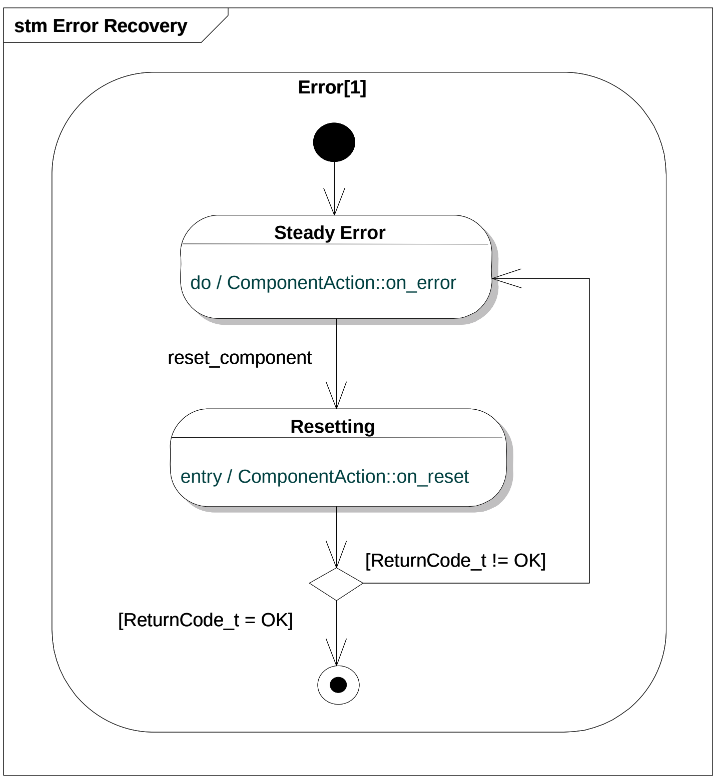

Figure 5.6 - Error Recovery

Constraints

- A class that realizes this interface must conform to the protocol state machine shown above.

Operations

`LightweightRTObject`
no attributes
operations

initialize		ReturnCode_t
finalize		ReturnCode_t
is_alive		Boolean
	context	ExecutionContext
exit		ReturnCode_t
attach_context		ExecutionContextHandle_t
	context	ExecutionContext
detach_context		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
get_context		ExecutionContext
	exec_handle	ExecutionContextHandle_t
get_owned_contexts		ExecutionContextOperations[]
get_participating_contexts		ExecutionContextOperations[]
get_context_handle		ExecutionContextHandle_t
	context	ExecutionContext

5.2.2.2.1 initialize

Description

Initialize the RTC that realizes this interface.

Semantics

The invocation of this operation shall result in the invocation of the callback **`ComponentAction::on_initialize`**.

Constraints

- An RTC may be initialized only while it is in the Created state. Any attempt to invoke this operation while in another state shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.
- Application developers are not expected to call this operation directly; it exists for use by the RTC infrastructure.

5.2.2.2.2 finalize

Description

Finalize the RTC that realizes this interface, preparing it for destruction.

Semantics

This invocation of this operation shall result in the invocation of the callback **`ComponentAction::on_finalize`**.

Constraints

- An RTC may not be finalized while it is participating in any execution context. It must first be removed with **`ExecutionContextOperations::remove_component`**. Otherwise, this operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**. See Figure 5.9.
- An RTC may not be finalized while it is in the Created state. Any attempt to invoke this operation while in that state shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.
- Application developers are not expected to call this operation directly; it exists for use by the RTC infrastructure.

5.2.2.2.3 is_alive

Description

A component is alive or not regardless of the execution context from which it is observed. However, whether or not it is Active, Inactive, or in Error is dependent on the execution context(s) (see Figure 5.7) in which it is running. That is, it may be Active in one context but Inactive in another. Therefore, this operation shall report whether this RTC is *either* Active, Inactive, or in Error; which of those states a component is in with respect to a particular context may be queried from the context itself.

5.2.2.2.4 exit

Description

Stop the RTC's execution context(s) and finalize it along with its contents.

Semantics

Any execution contexts for which the RTC is the **`owner`** shall be stopped.

If the RTC participates in any execution contexts belonging to another RTC that contains it, directly or indirectly (i.e., the containing RTC is the **`owner`** of the **`ExecutionContext`**), it shall be deactivated in those contexts.

After the RTC is no longer Active in any Running execution context, it and any RTCs contained transitively within it shall be finalized.

Constraints

- An RTC cannot be exited if it has not yet been initialized. Any attempt to exit an RTC that is in the Created state shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.

5.2.2.2.5 attach_context

Description

Inform this RTC that it is participating in the given execution context. Return a handle that represents the association of this RTC with the context.

Semantics

This operation is intended to be invoked by **`ExecutionContextOperations::add_component`** (see Section 5.2.2.6.6). It is not intended for use by other clients.

5.2.2.2.6 detach_context

Description

Inform this RTC that it is no longer participating in the given execution context.

Semantics

This operation is intended to be invoked by **`ExecutionContextOperations::remove_component`** (see Section 5.2.2.6.7). It is not intended for use by other clients.

Constraints

- This operation may not be invoked if this RTC is not already participating in the execution context. Such a call shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.
- This operation may not be invoked if this RTC is Active in the indicated execution context. Otherwise, it shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.

5.2.2.2.7 get_context

Description

Obtain a reference to the execution context represented by the given handle.

Semantics

The mapping from handle to context is specific to a particular RTC instance. The given handle must have been obtained by a previous call to **`attach_context`** on this RTC.

5.2.2.2.8 get_owned_contexts

Description

This operation returns a list of all execution contexts owned by this RTC.

5.2.2.2.9 get_participating_contexts

Description

This operation returns a list of all execution contexts in which this RTC participates.

Semantics

Each call to **`attach_context`** causes the provided context to be added to this list. Each call to **`detach_context`** causes the provided context to be removed from this list.

5.2.2.2.10 get_context_handle

Description

This operation returns a handle that is associated with the given execution context.

Semantics

The handle returned by this operation is same as the handle returned by attach_context.

5.2.2.3 LifeCycleState

Description

`LifeCycleState` is an enumeration of the states in the lifecycle above.

Attributes

LifeCycleState	
attributes	
`CREATED`	`LifeCycleState`
`INACTIVE`	`LifeCycleState`
`ACTIVE`	`LifeCycleState`
`ERROR`	`LifeCycleState`
no operations	

5.2.2.3.1 CREATED

Description

The RTC object has been instantiated but not yet fully initialized.

5.2.2.3.2 INACTIVE

Description

The RTC is Alive but is not being invoked in any execution context (see Section 5.2.2.5), regardless of whether the context is Running or not.

Semantics

An instance of this state exists for each execution context in which the RTC participates. If the RTC does not participate in any execution context, a single instance of this state exists.

5.2.2.3.3 ACTIVE

Description

The RTC is Alive and will be invoked in the execution context if the context is Running.

Semantics

An instance of this state exists for each execution context in which the RTC participates. If the RTC does not participate in any execution context, this state shall never be observed.

5.2.2.3.4 ERROR

Description

The RTC has encountered a problem in a given execution context and cannot continue functioning in that context without being reset.

5.2.2.4 ComponentAction

Description

The **ComponentAction** interface provides callbacks corresponding to the execution of the lifecycle operations of **LightweightRTObject** (see Section 5.2.2.2) and **ExecutionContext** (see Section 5.2.2.5). An RTC developer may implement these callback operations in order to execute application-specific logic pointing response to those transitions.

Semantics

Clients of an RTC are not expected to invoke these operations directly; they are provided for the benefit of the RTC middleware implementation.

Operations

ComponentAction		
no attributes		
operations		
on_initialize		ReturnCode_t
on_finalize		ReturnCode_t
on_startup		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
on_shutdown		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
on_activated		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
on_deactivated		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
on_aborting		ReturnCode_t
	exec_handle	ExecutionContextHandle_t
on_error		ReturnCode_t

	exec_handle	ExecutionContextHandle_t
on_reset		ReturnCode_t
	exec_handle	ExecutionContextHandle_t

5.2.2.4.1 on_initialize

Description

The RTC has been initialized and entered the Alive state.

Semantics

Any RTC-specific initialization logic should be performed here.

5.2.2.4.2 on_finalize

Description

The RTC is being destroyed.

Semantics

Any final RTC-specific tear-down logic should be performed here.

5.2.2.4.3 on_startup

Description

The given execution context, in which the RTC is participating, has transitioned from Stopped to Running.

5.2.2.4.4 on_shutdown

Description

The given execution context, in which the RTC is participating, has transitioned from Running to Stopped.

5.2.2.4.5 on_activated

Description

The RTC has been activated in the given execution context.

5.2.2.4.6 on_deactivated

Description

The RTC has been deactivated in the given execution context.

5.2.2.4.7 on_aborting

Description

The RTC is transitioning from the Active state to the Error state in some execution context.

Semantics

This callback is invoked only a single time for time that the RTC transitions into the Error state from another state. This behavior is in contrast to that of **`on_error`**.

5.2.2.4.8 on_error

Description

The RTC remains in the Error state.

Semantics

If the RTC is in the Error state relative to some execution context when it would otherwise be invoked from that context (according to the context's **`ExecutionKind`**), this callback shall be invoked instead. For example,

- If the **`ExecutionKind`** is **`PERIODIC`**, this operation shall be invoked in sorted order at the rate of the context instead of **`DataFlowComponentAction::on_execute`** and **`on_state_update`**.
- If the **`ExecutionKind`** is **`EVENT_DRIVEN`**, this operation shall be invoked whenever **`FsmParticipantAction::on_action`** would otherwise have been invoked.

5.2.2.4.9 on_reset

Description

The RTC is in the Error state. An attempt is being made to recover it such that it can return to the Inactive state.

Semantics

If the RTC was successfully recovered and can safely return to the Inactive state, this method shall complete with **`ReturnCode_t::OK`**. Any other result shall indicate that the RTC should remain in the Error state.

5.2.2.5 ExecutionContext

Description

An **`ExecutionContext`** allows the business logic of an RTC to be decoupled from the thread of control in which it is executed. The context represents a logical thread of control and is provided to RTCs at runtime as an argument to various operations, allowing them to query and modify their own state, and that of other RTCs executing within the same context, in the lifecycle.

This separation of concerns is important for two primary reasons:

- Large number of components may collaborate tightly within a single node or process. If each component were to run within its own thread of control, the infrastructure may not be able to satisfy the timeliness and determinism requirements of real-time applications due to the large number of threads and the required synchronization between them.

- A single application may carry out a number of independent tasks that require different execution rates. For example, it may need to sample a sensor periodically at a very high rate and update a user interface at a much lower rate.

Interface Realizations

- **`ExecutionContextOperations`**

Semantics

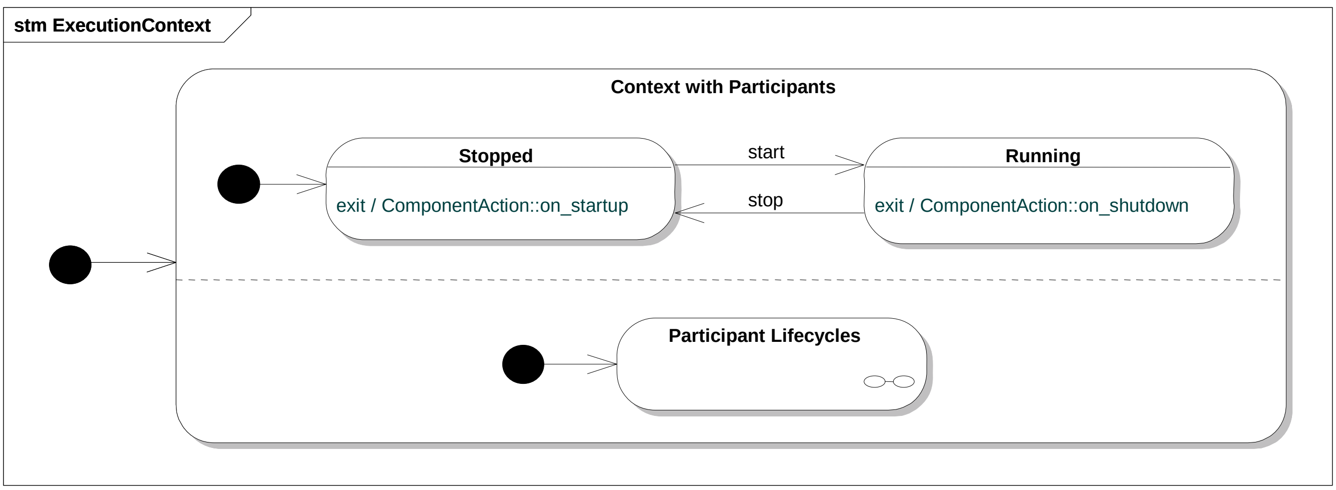

Figure 5.7 - ExecutionContext

The state machine of an **`ExecutionContext`** has two parts. The behavior of the **`ExecutionContext`** itself is defined by the upper region in the above figure. The behavior of the RTCs that participate in the context is defined by the lower region. The contents of that region are displayed in more detail in Figure 5.5 in Section 5.2.2.2.

Ownership and Participation

Each execution context is owned by a single RTC and may be used to execute that RTC and the RTCs contained within it, directly or indirectly. An RTC that owns one or more execution contexts is known as an *autonomous* RTC.

An autonomous RTC and some subset of the RTCs within it (to be defined by the application developer) shall be executed by the infrastructure according to the context's *execution kind*, which defines when each RTC's operations will be invoked when and in which order. These RTCs are said to *participate* in the context. The available execution kinds are described in Section 5.2.2.7.

The relationship between RTCs and execution contexts may be many-to-many in the general case: multiple RTCs may be invoked from the same execution context, and a single RTC may be invoked from multiple contexts. In the case where multiple RTCs are invoked from the same context, starting or stopping the context shall result in the corresponding lifecycle transitions for *all* of those components.

Logical and Physical Threads

Although an execution context represents a logical thread of control, the choice of how it maps to a physical thread shall be left to the application's deployment environment. Implementations may elect to associate contexts with threads with a one-to-one mapping, to serve multiple contexts from a single thread, or by any other means. In the case where a given RTC may be invoked from multiple contexts, concurrency management is implementation-dependent.

5.2.2.6 ExecutionContextOperations

Description

The **ExecutionContextOperations** interface defines the operations that all instances of **ExecutionContext** must provide.

Operations

ExecutionContextOperations		
no attributes		
operations		
is_running		Boolean
start		ReturnCode_t
stop		ReturnCode_t
get_rate		Double
set_rate		ReturnCode_t
	rate	Double
add_component		ReturnCode_t
	component	LightweightRTObject
remove_component		ReturnCode_t
	component	LightweightRTObject
activate_component		ReturnCode_t
	component	LightweightRTObject
deactivate_component		ReturnCode_t
	component	LightweightRTObject
reset_component		ReturnCode_t
	component	LightweightRTObject
get_component_state		LifeCycleState

	`component`	LightweightRTObject
`get_kind`		`ExecutionKind`

5.2.2.6.1 is_running

Description

This operation shall return true if the context is in the Running state.

Semantics

While the context is Running, all Active RTCs participating in the context shall be executed according to the context's execution kind.

5.2.2.6.2 start

Description

Request that the context enter the Running state. Once the state transition occurs, the **`ComponentAction::on_startup`** operation (see Section 5.2.2.4.3) will be invoked.

Semantics

An execution context may not be started until the RT components that participate in it have been initialized.

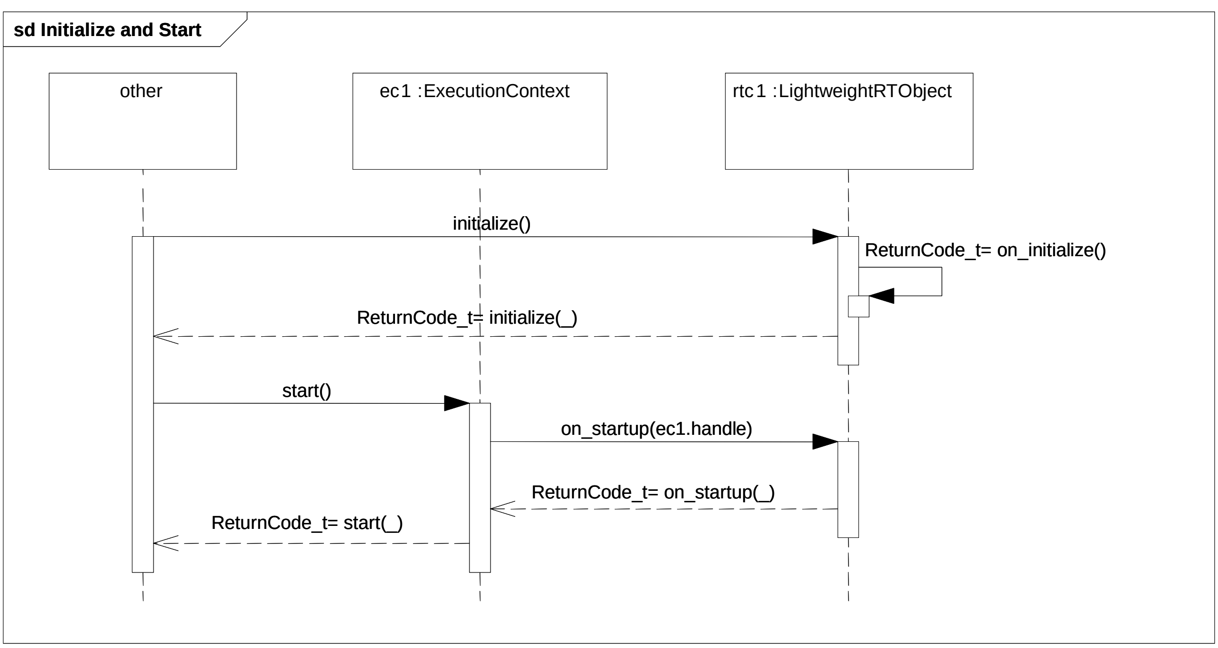

Figure 5.8 - Initialize and Start

An execution context may be started and stopped multiple times.

Constraints

- This operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`** if the context is not in the Stopped state.
- This operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`** if any of the participating components are not in their Alive state.

5.2.2.6.3 stop

Description

Request that the context enter the Stopped state. Once the transition occurs, the **`ComponentAction::on_shutdown`** operation (see Section 5.2.2.4.4) will be invoked.

Semantics

An execution context must be stopped before the RT components that participate in it are finalized.

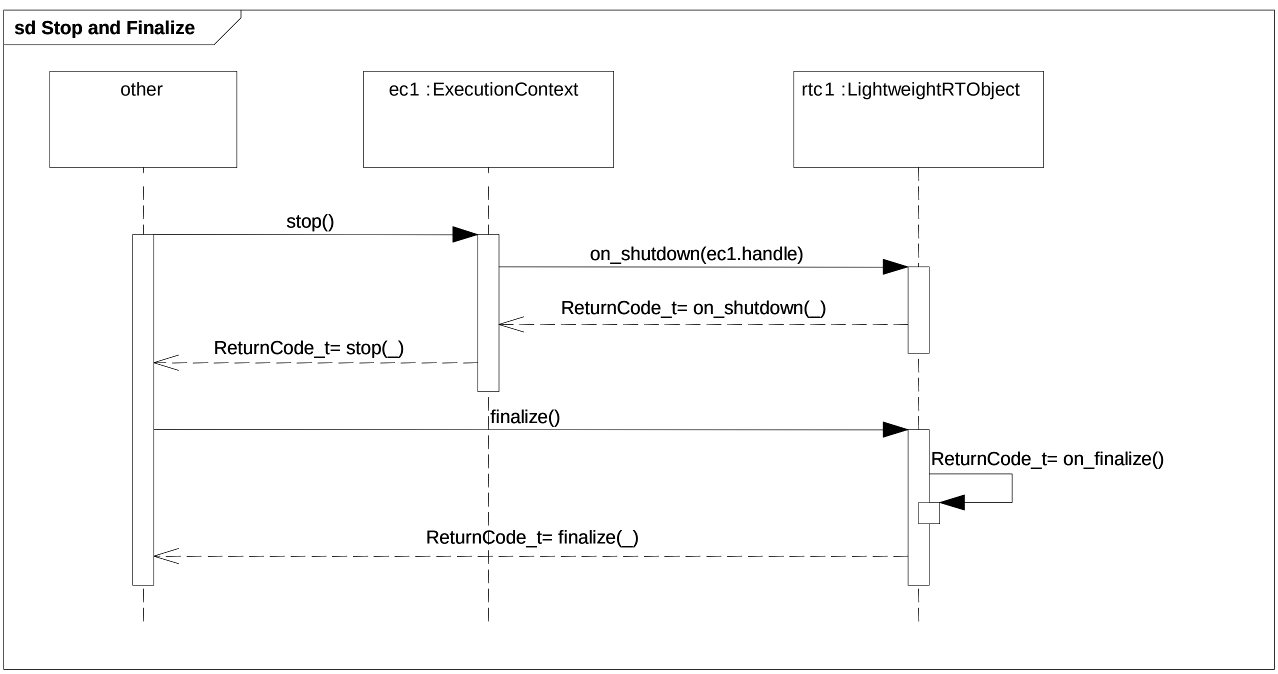

Figure 5.9 - Stop and Finalize

An execution context may be started and stopped multiple times.

Constraints

- This operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`** if the context is not in the Running state.

5.2.2.6.4 get_rate

Description

This operation shall return the rate (in hertz) at which its Active participating RTCs are being invoked.

Semantics

An implementation is permitted to perform some periodic or quasi-periodic processing within an execution context with an **ExecutionKind** other than **PERIODIC**. In such a case, the result of this operation is implementation-defined. If no periodic processing of any kind is taking place within the context, this operation shall fail as described in Section 5.2.1.

Constraints

- If the context has an **ExecutionKind** of **PERIODIC**, this operation shall return a rate greater than zero.

5.2.2.6.5 set_rate

Description

This operation shall set the rate (in hertz) at which this context's Active participating RTCs are being called.

Semantics

If the execution kind of the context is **PERIODIC**, a rate change shall result in the invocation of **on_rate_changed** on any RTCs realizing **DataFlowComponentAction** that are registered with any RTCs participating in the context.

An implementation is permitted to perform some periodic or quasi-periodic processing within an execution context with an **ExecutionKind** other than **PERIODIC**. If such is the case, and the implementation reports a rate from **get_rate**, this operation shall set that rate successfully provided that the given rate is valid. If no periodic processing of any kind is taking place within the context, this operation shall fail with **ReturnCode_t::UNSUPPORTED**.

Constraints

- The given rate must be greater than zero. Otherwise, this operation shall fail with **ReturnCode_t::BAD_PARAMETER**.

5.2.2.6.6 add_component

Description

The operation causes the given RTC to begin participating in the execution context.

Semantics

The newly added RTC will receive a call to **LightweightRTComponent::attach_context** (see Section 5.2.2.2.5) and then enter the Inactive state.

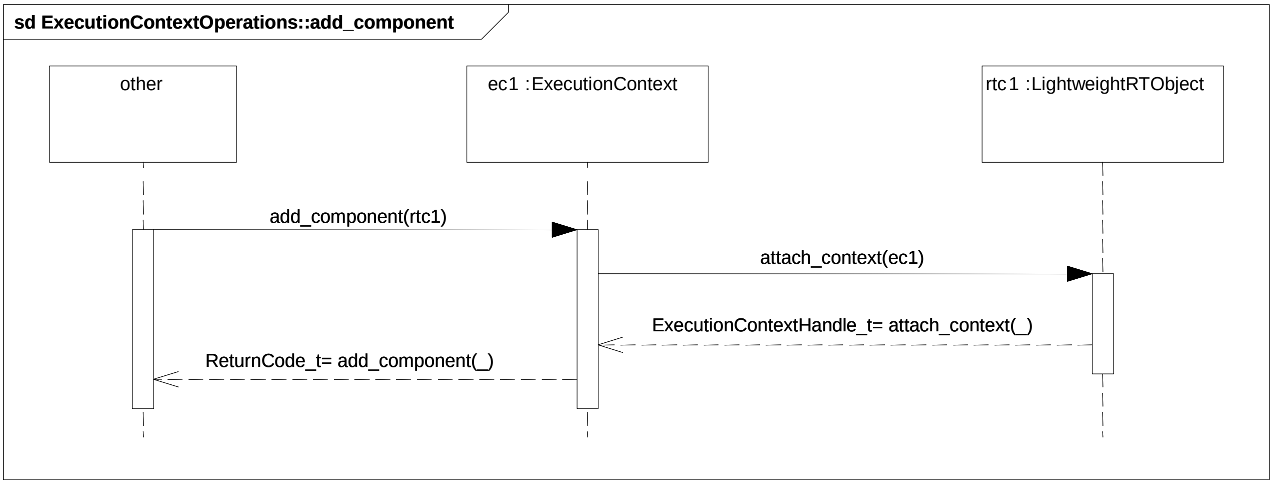

Figure 5.10 - ExecutionContextOperations::add_component

Constraints

- If the ExecutionKind of this context is **PERIODIC**, the RTC must be a data flow component (see Section 5.3.1.1). Otherwise, this operation shall fail with **ReturnCode_t::PRECONDITION_NOT_MET**.
- If the ExecutionKind of this context is **EVENT_DRIVEN**, the RTC must be an FSM participant (see Section 5.3.2.3). Otherwise, this operation shall fail with **ReturnCode_t::PRECONDITION_NOT_MET**.

5.2.2.6.7 remove_component

Description

This operation causes a participant RTC to stop participating in the execution context.

Semantics

The removed RTC will receive a call to **LightweightRTComponent::detach_context** (see Section 5.2.2.2.6).

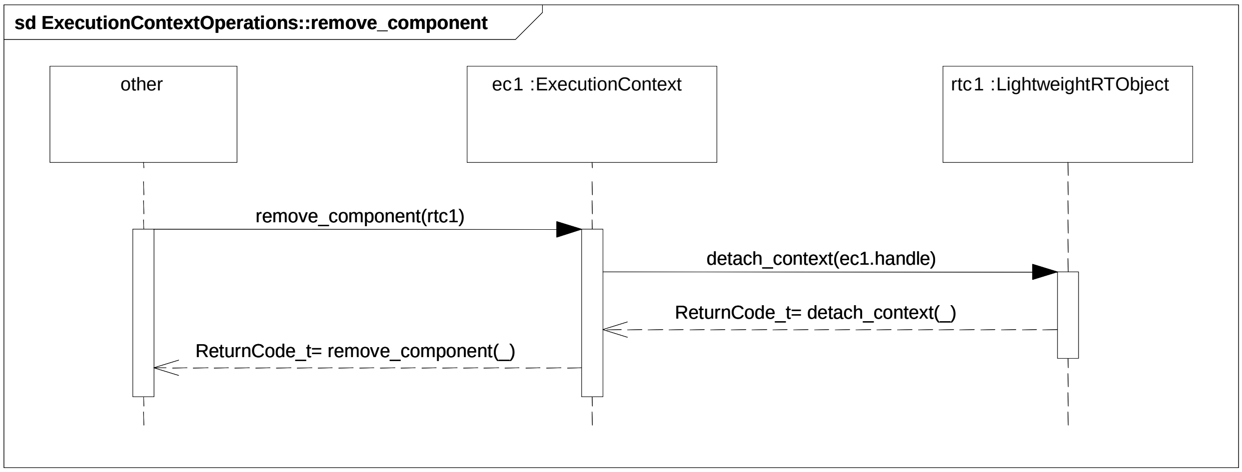

Figure 5.11 - ExecutionContextOperations::remove_component

Constraints

- If the given RTC is not currently participating in the execution context, this operation shall fail with **`ReturnCode_t::BAD_PARAMETER`**.
- An RTC must be deactivated before it can be removed from an execution context. If the given RTC is participating in the execution context but is still in the Active state, this operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.

5.2.2.6.8 activate_component

Description

The given participant RTC is Inactive and is therefore not being invoked according to the execution context's execution kind. This operation shall cause the RTC to transition to the Active state such that it may subsequently be invoked in this execution context.

Semantics

The callback **`on_activate`** shall be called as a result of calling this operation. This operation shall not return until the callback has returned, and shall result in an error if the callback does.

The following figure is a non-normative example sequence diagram for **`activate_component`**.

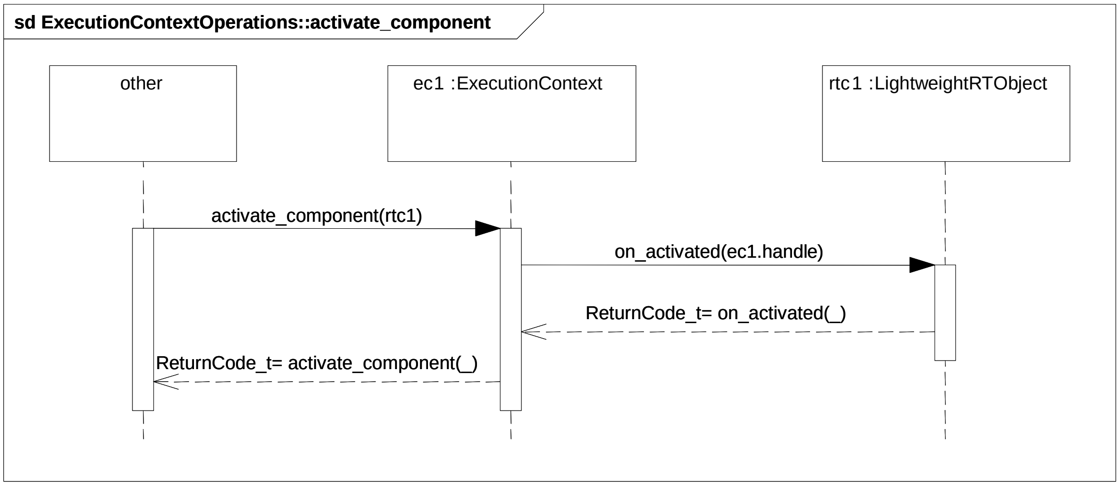

Figure 5.12 - ExecutionContextOperations::activate_component

Constraints

- An execution context can only activate its participant components. If the given RTC is not participating in the execution context, this operation shall fail with **`ReturnCode_t::BAD_PARAMETER`**.
- An RTC that is in the Error state cannot be activated until after it has been reset. If the given RTC is in the Error state, this operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**.
- This operation shall fail with **`ReturnCode_t::BAD_PARAMETER`** if the given component is not in its Alive state.

5.2.2.6.9 deactivate_component

Description

The given RTC is Active in the execution context. Cause it to transition to the Inactive state such that it will not be subsequently invoked from the context unless and until it is activated again.

Semantics

The callback **`on_deactivate`** shall be called as a result of calling this operation. This operation shall not return until the callback has returned, and shall result in an error if the callback does.

The following figure is a non-normative example sequence diagram for **`deactivate_component`**.

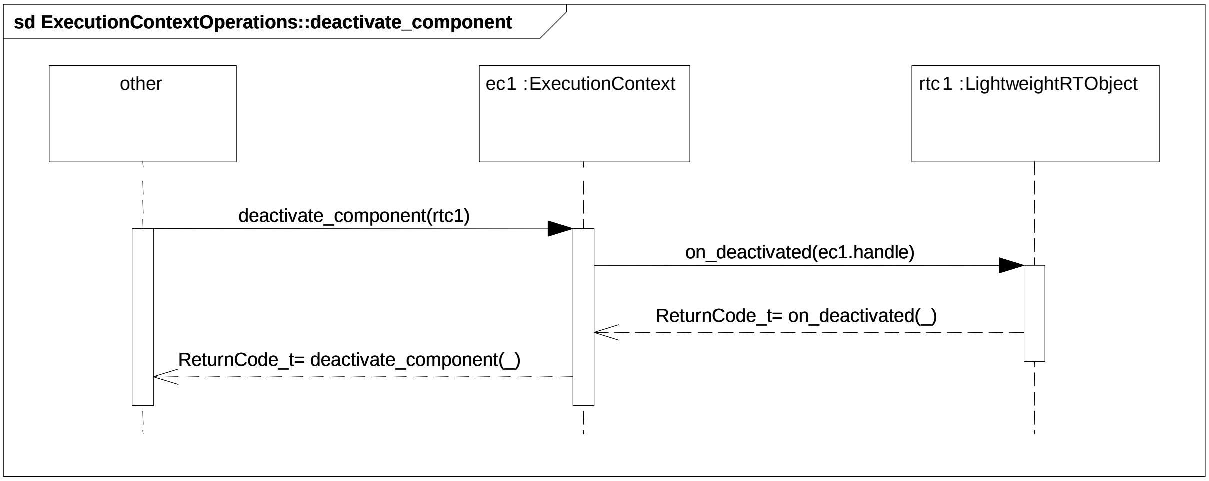

Figure 5.13 - ExecutionContextOperations::deactivate_component

Constraints

- An execution context can only deactivate its participant components. If the given RTC is not participating in the execution context, this operation shall fail with **`ReturnCode_t::BAD_PARAMETER`**.
- This operation shall fail with **`ReturnCode_t::BAD_PARAMETER`** if the given component is not in its Alive state.

5.2.2.6.10 reset_component

Description

Attempt to recover the RTC when it is in Error.

Semantics

The **`ComponentAction::on_reset`** callback shall be invoked. This operation shall not return until the callback has returned, and shall result in an error if the callback does. If possible, the RTC developer should implement that callback such that the RTC may be returned to a valid state.

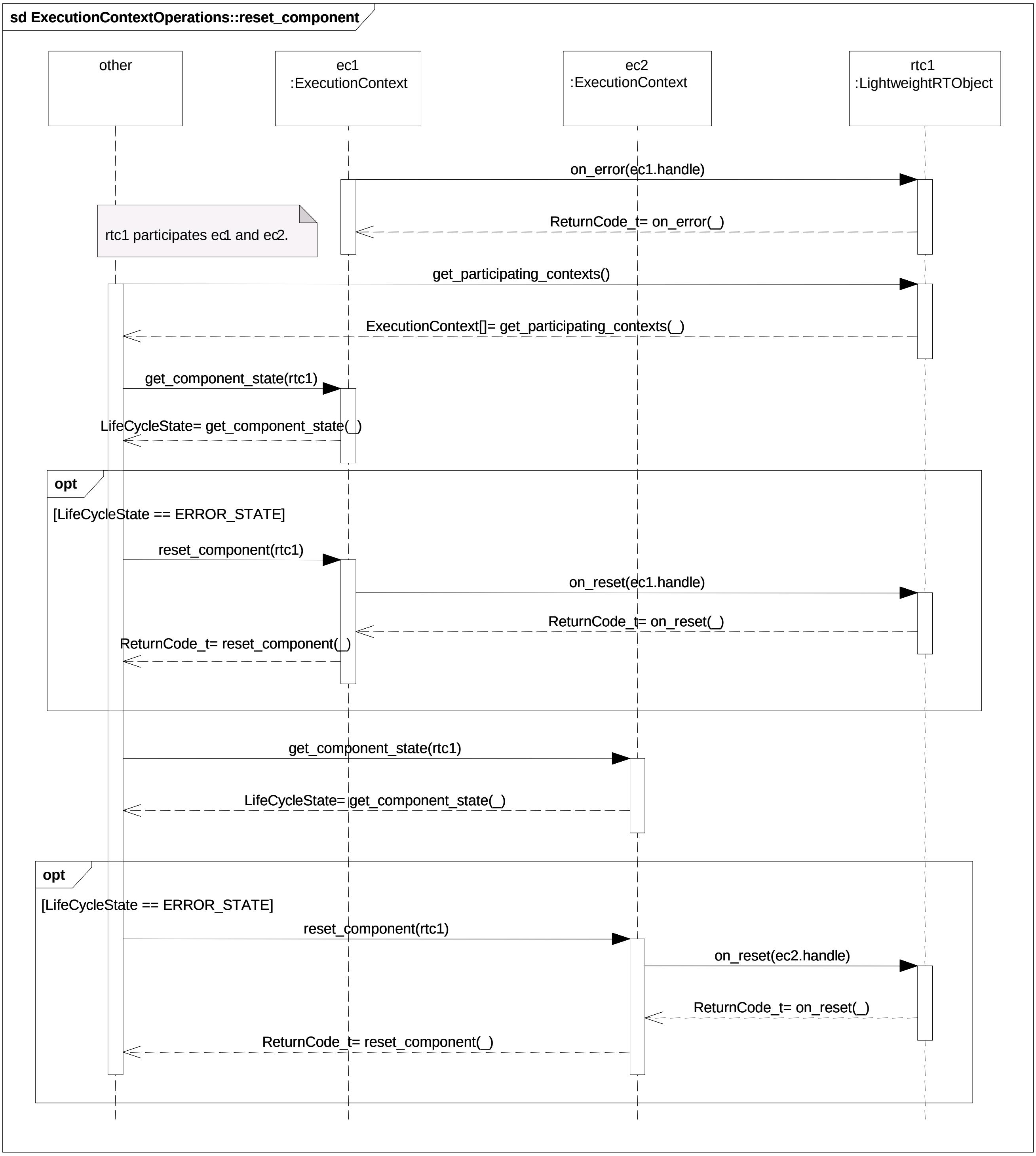

Figure 5.14 - ExecutionContextOperations::reset_component

If this operation fails, the RTC will remain in Error.

Constraints

- An RTC may only be reset in an execution context in which it is in error. If the RTC is not in Error in the identified context, this operation shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**. However, that failure shall not cause the RTC to enter the Error state.
- An RTC may not be reset while in the Created state. Any attempt to invoke this operation while the RTC is in that state shall fail with **`ReturnCode_t::PRECONDITION_NOT_MET`**. However, that failure shall not cause the RTC to enter the Error state.

5.2.2.6.11 get_component_state

Description

This operation shall report the **`LifeCycleState`** of the given participant RTC.

Constraints

- The given RTC must be Alive.
- The given RTC must be a participant in the target execution context.
- The **`LifeCycleState`** returned by this operation shall be one of **`LifeCycleState::INACTIVE`**, **`ACTIVE`**, or **`ERROR`**.

5.2.2.6.12 get_kind

Description

This operation shall report the execution kind of the execution context.

5.2.2.7 ExecutionKind

Description

The **`ExecutionKind`** enumeration defines the execution semantics (see Section 5.3) of the RTCs that participate in an execution context.

Attributes

ExecutionKind	
attributes	
PERIODIC	ExecutionKind
EVENT_DRIVEN	ExecutionKind
OTHER	ExecutionKind
no operations	

5.2.2.7.1 PERIODIC

Description

The participant RTCs are executing according to periodic sampled data semantics (see Section 5.3.1).

5.2.2.7.2 EVENT_DRIVEN

Description

The participant RTCs are executing according to stimulus response semantics (see Section 5.3.2).

5.2.2.7.3 OTHER

Description

The participant RTCs are executing according to some semantics not defined by this specification.

5.2.2.8 ExecutionContextHandle_t

Description

This data type represents the association between an RTC and an ExecutionContext in which it participates.

Semantics

This is an opaque DataType. It has no attributes or operations.

5.2.3 Basic Types

This specification reuses the primitive types from [UML]: String, Boolean, etc. It also defines additional primitive types that are not available from UML, such as types for representing floating point numbers[2]. These additional types are described in this section.

5.2.3.1 Character

Description

The **`Character`** primitive type is an 8-bit quantity that encodes a single-byte character from any byte-oriented code set.

`Character` is an instance of **`PrimitiveType`** [UML].

Constraints

The **`Character`** value is a **`LiteralCharacter`** [UML].

5.2.3.2 Double

Description

The **`Double`** primitive type is an IEEE double-precision floating point number. See IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, for a detailed specification.

`Double` is an instance of **`PrimitiveType`** [UML].

Constraints

The **`Double`** value is a **`LiteralDouble`**.

5.2.3.3 Float

Description

The **`Float`** primitive type is an IEEE single-precision floating point number. See IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, for a detailed specification.

`Float` is an instance of **`PrimitiveType`** [UML].

Constraints

The **`Float`** value shall be a **`LiteralFloat`** that is an IEEE single-precision floating point number.

---

2. *Note on model reuse*: Models of the IDL basic types are also defined in the UML Profile for CORBA Specification [CORBAUML] and in the PIM and PSM for Software Radio Components document [SWRADIO] (although those two definitions differ from one another). CORBA and Software Radio Components are considered to be platforms by this specification; it is not appropriate for this PIM to depend on either of those specifications. Therefore, this specification defines the necessary types itself.

5.2.3.4 Long

Description

The **Long** primitive type, a specialization of **Integer** primitive type, is a signed integer range [-2^{31}, $2^{31} - 1$].

Long is an instance of **PrimitiveType** [UML].

Constraints

The **Long** value shall be a **LiteralInteger** [UML] with a range of [-2^{31}, $2^{31} - 1$].

5.2.3.5 LongDouble

Description

The **LongDouble** primitive type is an IEEE double-extended floating-point number. See IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, for a detailed specification.

LongDouble is an instance of **PrimitiveType** [UML].

Constraints

The **LongDouble** value shall be a **LiteralLongDouble** that is an IEEE double-extended floating-point number.

5.2.3.6 LongLong

Description

The **LongLong** primitive type, a specialization of **Integer** primitive type, is a signed integer range [-2^{63}, $2^{63} - 1$].

LongLong is an instance of **PrimitiveType** [UML].

Constraints

The **LongLong** value shall be a **LiteralInteger** [UML] with a range of [-2^{63}, $2^{63} - 1$].

5.2.3.7 Octet

Description

The **Octet** primitive type, a specialization of Integer primitive type, is an unsigned integer within range [0, 255].

Octet is an instance of **PrimitiveType** [UML].

Constraints

The **Octet** value shall be a **LiteralInteger** [UML] with a range of [0, 255].

5.2.3.8 Short

Description

The **Short** primitive type, a specialization of **Integer** primitive type, is a signed integer range [-2^{15}, $2^{15} - 1$].

Short is an instance of **PrimitiveType** [UML].

Constraints

The **Short** value shall be a **LiteralInteger** [UML] with a range of [-2^{15}, $2^{15} - 1$].

5.2.3.9 UnsignedLong

Description

The **UnsignedLong** primitive type, a specialization of **Integer** primitive type, is an unsigned integer range [0, $2^{32} - 1$].

UnsignedLong is an instance of **PrimitiveType** [UML].

Constraints

The **UnsignedLong** value shall be a **LiteralInteger** [UML] with a range of [0, $2^{32} - 1$].

5.2.3.10 UnsignedLongLong

Description

The **UnsignedLongLong** primitive type, a specialization of **Integer** primitive type, is an unsigned integer range [0, $2^{64} - 1$].

UnsignedLongLong is an instance of **PrimitiveType** [UML].

Constraints

The **UnsignedLongLong** value shall be a **LiteralInteger** [UML] with a range of [0, $2^{64} - 1$].

5.2.3.11 UnsignedShort

Description

The **UnsignedShort** primitive type, a specialization of **Integer** primitive type, is an unsigned integer range [0, $2^{16} - 1$].

UnsignedShort is an instance of **PrimitiveType** [UML].

Constraints

The **UnsignedShort** value shall be a **LiteralInteger** [UML] with a range of [0, $2^{16} - 1$].

5.2.3.12 WideCharacter

Description

The **WideCharacter** primitive type represents a wide character that can be used for any character set.

WideCharacter is an instance of **PrimitiveType** [UML].

Constraints

The **`WideCharacter`** value shall be a **`LiteralWideCharacter`**.

5.2.3.13 WideString

Description

The **`WideString`** primitive type represents a wide character string that can be used for any character set.

`WideString` is an instance of **`PrimitiveType`** [UML].

Constraints

The **`WideString`** value is a **`LiteralWideString`**.

5.2.4 Literal Specifications

From [UML]:

> A literal specification is an abstract specialization of **`ValueSpecification`** that identifies a literal constant being modeled.

5.2.4.1 LiteralCharacter

Description

A literal character contains a **`Character`**-valued attribute.

Generalizations

- **`LiteralSpecification`** [UML]

Attributes

- value : Character

Semantics

A **`LiteralCharacter`** specifies a constant **`Character`** value.

5.2.4.2 LiteralDouble

Description

A literal double contains a **`Double`**-valued attribute.

Generalizations

- **`LiteralSpecification`** [UML]

Attributes

- value : Double

Semantics

A **LiteralDouble** specifies a constant **Double** value.

5.2.4.3 LiteralFloat

Description

A literal float contains a **Float**-valued attribute.

Generalizations

- **LiteralSpecification** [UML]

Attributes

- value : Float

Semantics

A **LiteralFloat** specifies a constant **Float** value.

5.2.4.4 LiteralLongDouble

Description

A literal long double contains a **LongDouble**-valued attribute.

Generalizations

- **LiteralSpecification** [UML]

Attributes

- value : LongDouble

Semantics

A **LiteralLongDouble** specifies a constant **LongDouble** value.

5.2.4.5 LiteralWideCharacter

Description

A literal wide character contains a **WideCharacter**-valued attribute.

Generalizations

- **LiteralSpecification** [UML]

Attributes

- value : WideCharacter

Semantics

A **`LiteralWideCharacter`** specifies a constant wide character value.

5.2.4.6 LiteralWideString

Description

A literal wide string contains a **`WideString`**-valued attribute.

Generalizations

- **`LiteralSpecification`** [UML]

Attributes

- value : WideString

Semantics

A **`LiteralWString`** specifies a constant wide string value.

5.3 Execution Semantics

Applications in many domains may benefit from a component-oriented design. Complex distributed control applications—including robotics applications—tend to share certain fundamental design patterns. Chief among them are periodic sampled data processing (also known as *data flow*), stimulus response processing (also known as *asynchronous* or *discrete event processing*), and modes of operation. This specification provides direct support for these design patterns in the sections that follow.

The semantics described in this section are applied to RTCs by means of stereotypes. These stereotypes are orthogonal and may be combined as desired within a single RTC. For example, a single RTC may participate in both periodic and stimulus response processing, or may execute periodically in multiple modes.

5.3.1 Periodic Sampled Data Processing

Real-time applications are frequently implemented using a methodology known as *periodic sampled data processing* or sometimes simply *data flow*[3]. This section defines the support for that design pattern and corresponds to the **`PERIODIC`** execution kind (see Section 5.2.2.7.1). A periodic execution context executes data flow components periodically in a well-defined order relative to one another. A data flow component may also be a composite component containing other data flow components. In this way, sampled data processing may be decomposed to an arbitrary level of hierarchy.

---

3. This methodology is supported by a number of existing software tools for real-time algorithm and/or application development, including Simulink from The MathWorks, LabView from National Instruments, and Constellation from RTI.

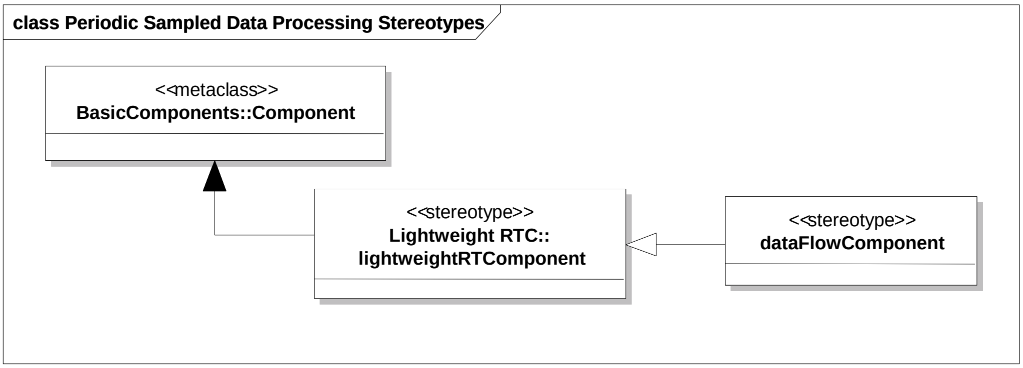

Figure 5.15 - Periodic Sampled Data Processing Stereotypes

An RTC developer declares a particular RTC to be a data flow component by extending it with the stereotype **`dataFlowComponent`**. The following figure is a non-normative example.

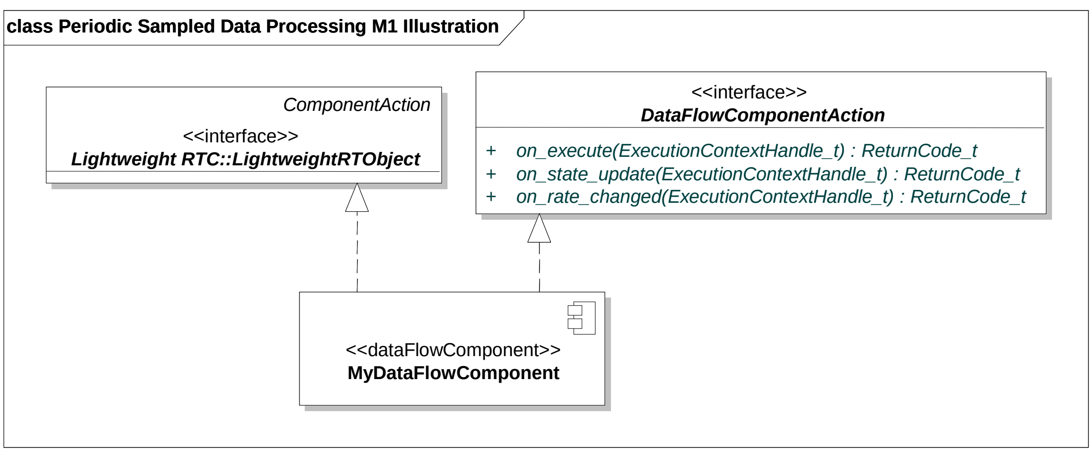

Figure 5.16 - Periodic Sampled Data Processing M1 Illustration

5.3.1.1 dataFlowComponent

Description

The **`dataFlowComponent`** stereotype may be applied to a component type to indicate that its instances should be executed in sorted order by a periodic execution context.

Constraints

- An instance of a component extended by the **`dataFlowComponent`** stereotype must participate in at least one execution context of kind **`PERIODIC`**, which shall also be used for the execution of any contained data flow components.
- A component extended by **`dataFlowComponent`** must realize the interface **`DataFlowComponentAction`**.

Generalizations

- `lightweightRTComponent`

5.3.1.1.1 Execution Sorting

A data flow component RTC determines the order in which to execute its contained data flow component RTCs by examining their interconnections in a process called *sorting*. The rules of sorting are simple: an RTC instance that produces an output (i.e., provides an interface) must run before any RTC instance that consumes that output (i.e., requires that interface). For example, if a provided interface of RTC instance A is connected to a required interface of RTC instance B, then A must be executed before B.

A data flow component RTC may contain component instances that are not data flow components. It shall ignore those instances for the purposes of periodic execution. However, other contained component instances may still invoke operations on the non-participant component instances if they choose.

Figure 5.17 depicts a non-normative example of a data flow component RTC with data flow components inside. The participants are executed in the order shown based upon their interconnections. The contained component that is not a data flow component is executed only when explicitly called by a data flow component.

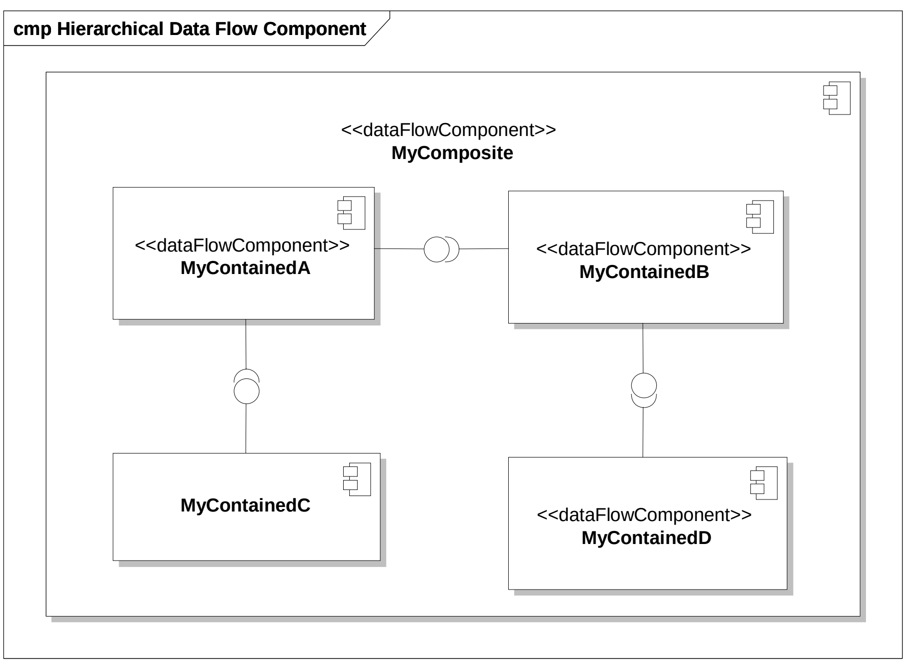

Figure 5.17 - Hierarchical Data flow Component

5.3.1.1.2 Two-Pass Execution

In real-time applications, the need to minimize latency and jitter is critical. To reduce latency, the execution of each leaf RTC should occur as closely as possible to the beginning of the sample period. To reduce jitter, the execution of a *particular* leaf RTC should always occur *at the same time* relative to the beginning of the sample period. For these reasons, it is undesirable for an RTC to perform any task whose running time is long or indeterminate within **`on_execute`**.

Furthermore, it is common for several collaborating RTCs to share access to a single data value or other resource. These RTCs should observe this resource in a consistent state during any sample period; i.e., it is inappropriate for any RTC to change the state of the resource before the other RTCs have executed.

In order to support the use cases just described (i.e., low latency, low jitter, and shared state), this specification supports two-pass execution of data flow component RTCs. In the first pass, each participant RTC's **`on_execute`** (see Section 5.3.1.2.1) operation shall be invoked in sorted order. After the completion of the first pass, each RTC's **`on_state_update`** (see Section 5.3.1.2.2) operation shall be invoked in sorted order. For example, in Figure 5.17 the order of calls would be:

1. MyContainedA.on_execute()
2. MyContainedB.on_execute()
3. MyContainedD.on_execute()
4. MyContainedA.on_state_update()
5. MyContainedB.on_state_update()
6. MyContainedD.on_state_update()

A data flow component should perform its primary business logic in **`on_execute`** but delay any expensive operations or changes to shared state until **`on_state_update`**.

The following figures are non-normative example sequence diagrams for two-pass execution and **`ExecutionContextOperations::set_rate`**.

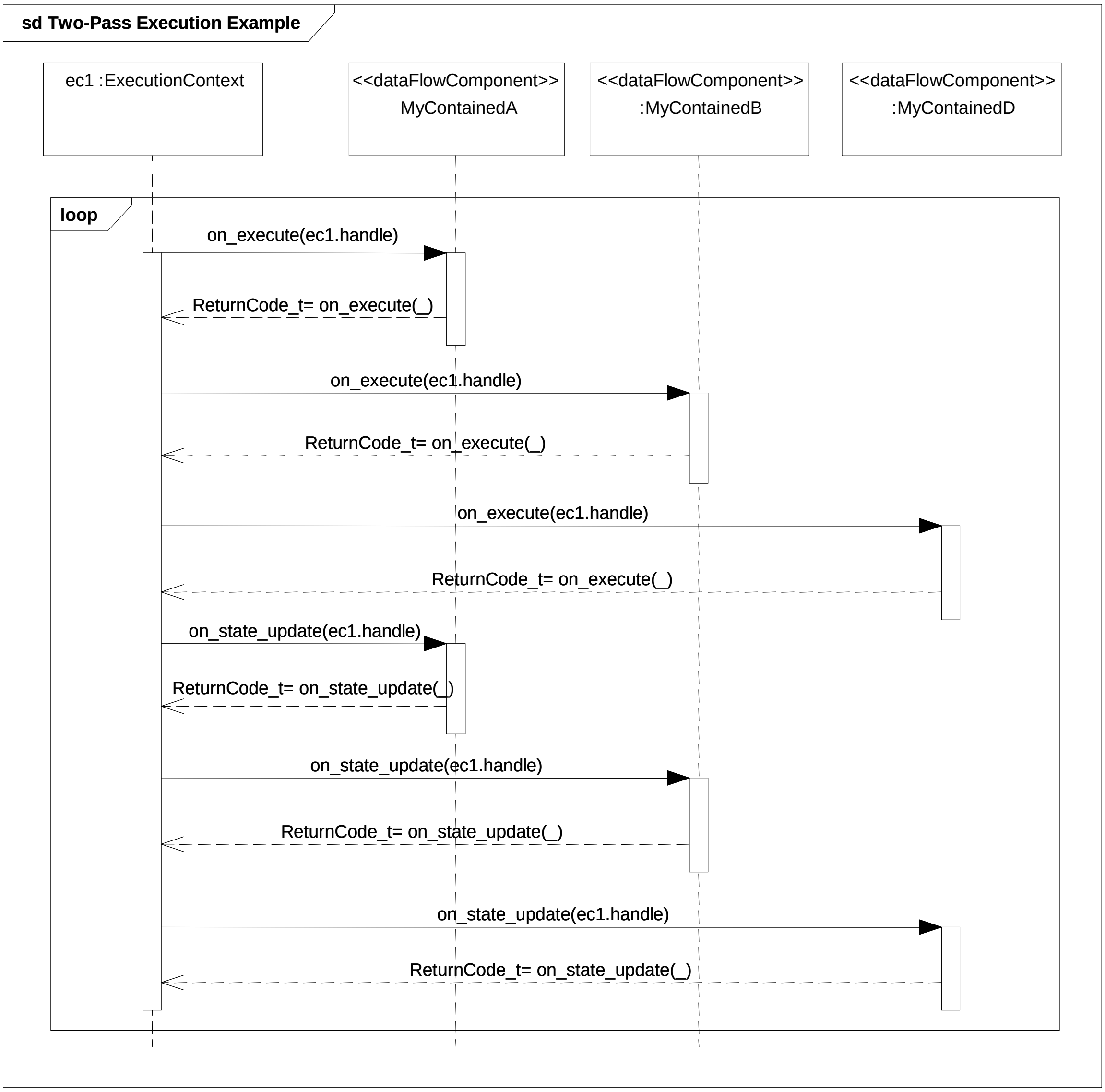

Figure 5.18 - Two Pass Execution Example

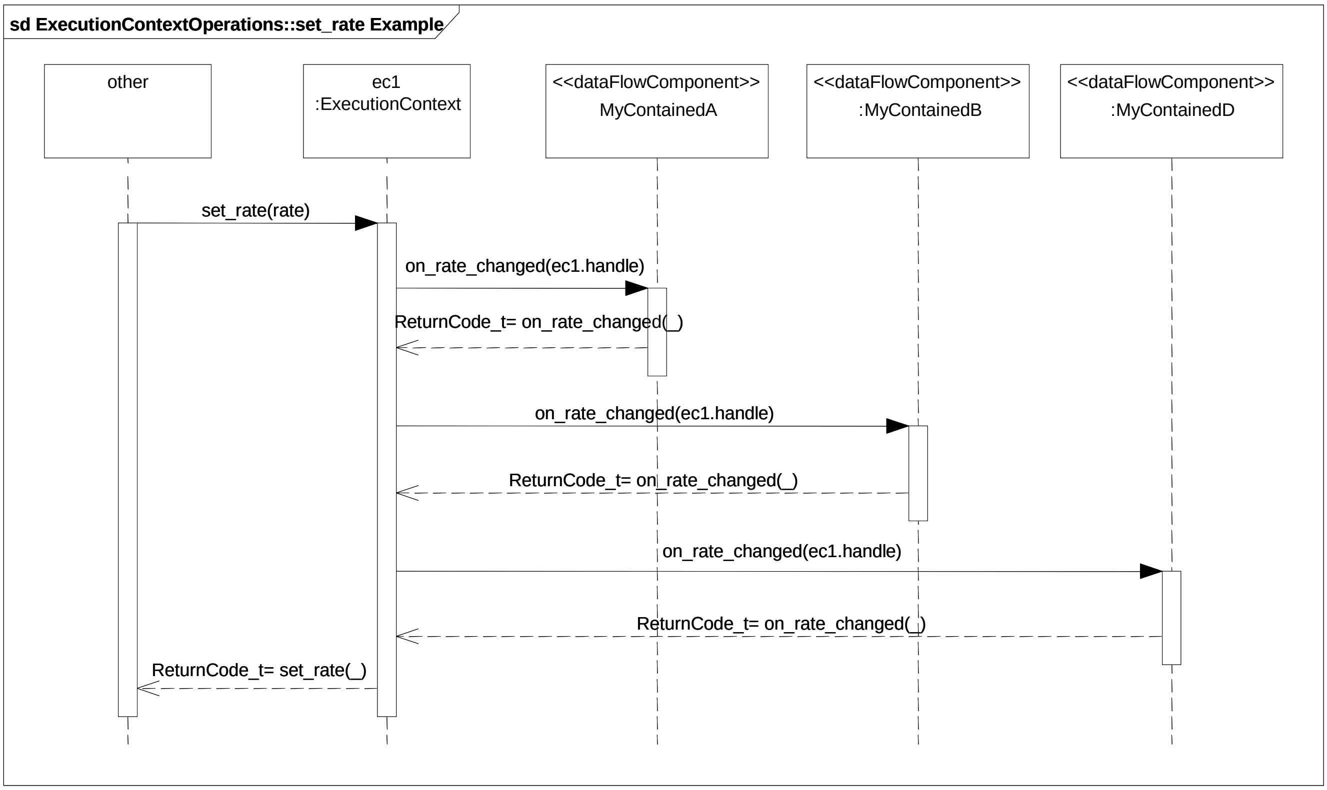

Figure 5.19 - ExecutionContextOperations::set_rate Example

5.3.1.2 DataFlowComponentAction

Description

`DataFlowComponentAction` is a companion to **`ComponentAction`** (see Section 5.2.2.4) that provides additional callbacks for intercepting the two execution passes defined in Section 5.3.1.1.2.

Operations

`DataFlowComponentAction`		
no attributes		
operations		
`on_execute`		`ReturnCode_t`
	`exec_handle`	ExecutionContextHandle_t
`on_state_update`		`ReturnCode_t`
	`exec_handle`	ExecutionContextHandle_t

on_rate_changed		ReturnCode_t
	exec_handle	ExecutionContextHandle_t

5.3.1.2.1 on_execute

Description

This operation will be invoked periodically at the rate of the given execution context as long as the following conditions hold:

- The RTC is Active.
- The given execution context is Running.

Semantics

This callback occurs during the first execution pass.

Constraints

- The execution context of the given context shall be **PERIODIC**.

5.3.1.2.2 on_state_update

Description

This operation will be invoked periodically at the rate of the given execution context as long as the following conditions hold:

- The RTC is Active.
- The given execution context is Running.

Semantics

This callback occurs during the second execution pass.

Constraints

- The execution context of the given context shall be **PERIODIC**.

5.3.1.2.3 on_rate_changed

Description

This operation is a notification that the rate of the indicated execution context (see Section 5.2.2.6.4) has changed.

Constraints

- The execution context of the given context shall be **PERIODIC**.

5.3.2 Stimulus Response Processing

The *stimulus response* design pattern is also referred to as *asynchronous* or *discrete event processing*. Applications use this pattern when they need to respond asynchronously to changes in their environment. The behavior is modeled using finite state machines. The state machine waits until the asynchronous arrival of an "event"[4], at which point it wakes up and transitions to a new state, executing an action associated with that transition.

This design pattern corresponds to the **`EVENT_DRIVEN`** execution kind (see Section 5.2.2.7.2).

Stimulus response consists of two roles:

- A *finite state machine* composite component (FSM) contains a set of named states and transitions between them. Each transition, as well as the entry and exit of each state, may be associated with the execution of a contained finite state machine participant instance.
- A *finite state machine participant* declares to a containing FSM that it is available for execution when a transition is taken or when a state is entered or exited.

In many cases, the FSM will also be an FSM participant, allowing it to appear as a submachine state within another FSM.

Stimulus response models can be inspected and modified dynamically using types in the Introspection package. The correspondence between state machine transitions, state entries, and state exits in an FSM and FSM participants is described in an **`FsmProfile`** (see Section 5.4.1.7) and its **`FsmBehaviorProfiles`** (see Section 5.4.1.8) in the Introspection package (see Section 5.4). This data is provided by an **`FsmService`** interface (see Section 5.4.2.5) in the Introspection package.

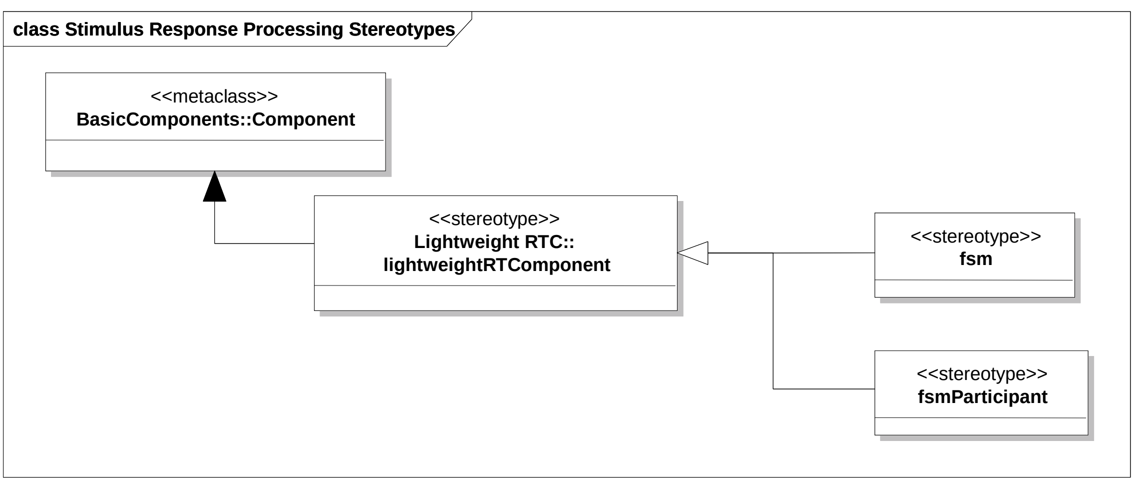

Figure 5.20 - Stimulus Response Processing Stereotypes

The RTC developer declares a particular RTC to be an FSM, or FSM participant, by extending it with the stereotype **`fsm`** (see Section 5.3.2.1) or **`fsmParticipant`** (see Section 5.3.2.3) respectively.

4. An Event in the nomenclature of finite state machines [UML] is not to be confused with "events" as understood within the context of the CORBA Event Service. Any relationship between the two is implementation-defined.

The following figure is a non-normative example of a UML model that incorporates these stereotypes.

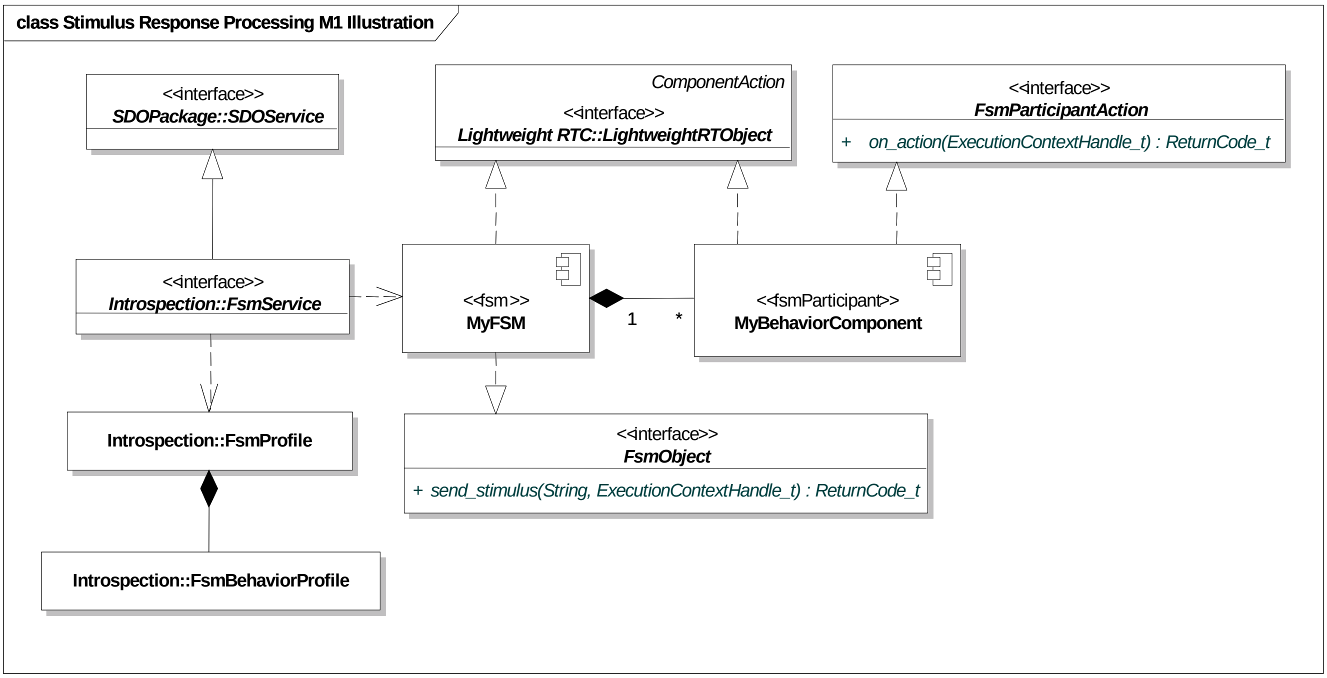

Figure 5.21 - Stimulus Response Processing M1 Illustration

5.3.2.1 fsm

Description

Applying the **`fsm`** stereotype to a component implies the ability to define component-specific states and transitions.

Semantics

In creating a state machine such as is depicted in Figure 5.22, the RTC developer is implicitly defining the Active state to be a submachine state.

The **`BehaviorStateMachines`** package described in [UML] is considered the normative definition of a state machine.

Generalizations

- **`lightweightRTComponent`**

Constraints

- A component extended by **`fsm`** must realize the interface **`FsmObject`**.
- A component extended by the **`fsm`** stereotype must participate in at least one execution context of kind **`EVENT_DRIVEN`**.
- The **`isSubmachineState`** attribute of the Active state must be **`true`**.

- The **submachine** association of the Active state must refer to a non-nil **StateMachine**.

5.3.2.2 FsmObject

Description

The **FsmObject** interface allows programs to send stimuli to a finite state machine, possibly causing it to change states.

Operations

FsmObject		
no attributes		
operations		
send_stimulus		ReturnCode_t
	message	String
	exec_handle	ExecutionContextHandle_t

5.3.2.2.1 send_stimulus

Description

Send a stimulus to an FSM that realizes this interface.

Semantics

If the stimulus corresponds to any outgoing transition of the current state, that transition shall be taken and the state shall change. Any FSM participants associated with the exit of the current state, the transition to the new state, or the entry to the new state shall be invoked. If the stimulus does not correspond to any such transition, this operation shall succeed but have no effect.

If the given execution context is a non-nil reference to a context in which this FSM participates, the transition shall be executed in that context. If the argument is nil, the FSM shall choose an **EVENT_DRIVEN** context in which to execute the transition. If the argument is non-nil, but this FSM does not participate in the given context, this operation shall fail with **ReturnCode_t::BAD_PARAMETER**.

Constraints

- The given execution context shall be of kind **EVENT_DRIVEN**.

5.3.2.3 fsmParticipant

Description

Applying the **fsmParticipant** stereotype to a component implicitly defines a **Behavior** [UML] of the same name as the RTC instance itself that may be invoked by a containing FSM when transitioning between states, entering or exiting a state, and so forth. The invocation of the **Behavior** corresponds to the invocation of the **on_action** (see Section 5.3.2.4.1) operation of the FSM participant RTC.

Figure 5.22 is a non-normative example of how a **`Behavior`** defined in terms of an FSM participant might be displayed graphically.

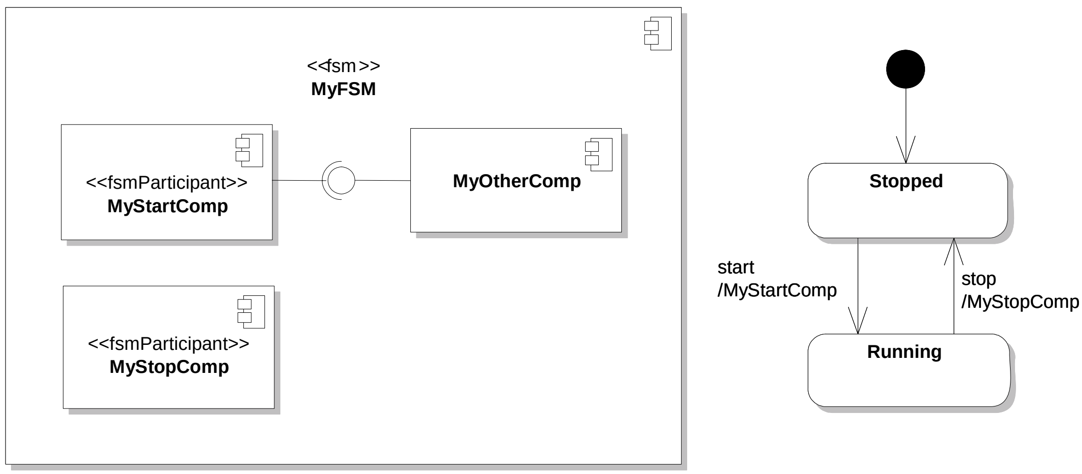

Figure 5.22 - FSM Participant Defines State Transition Behavior

Semantics

The following figure is a non-normative example sequence diagram for Stimulus Response Processing.

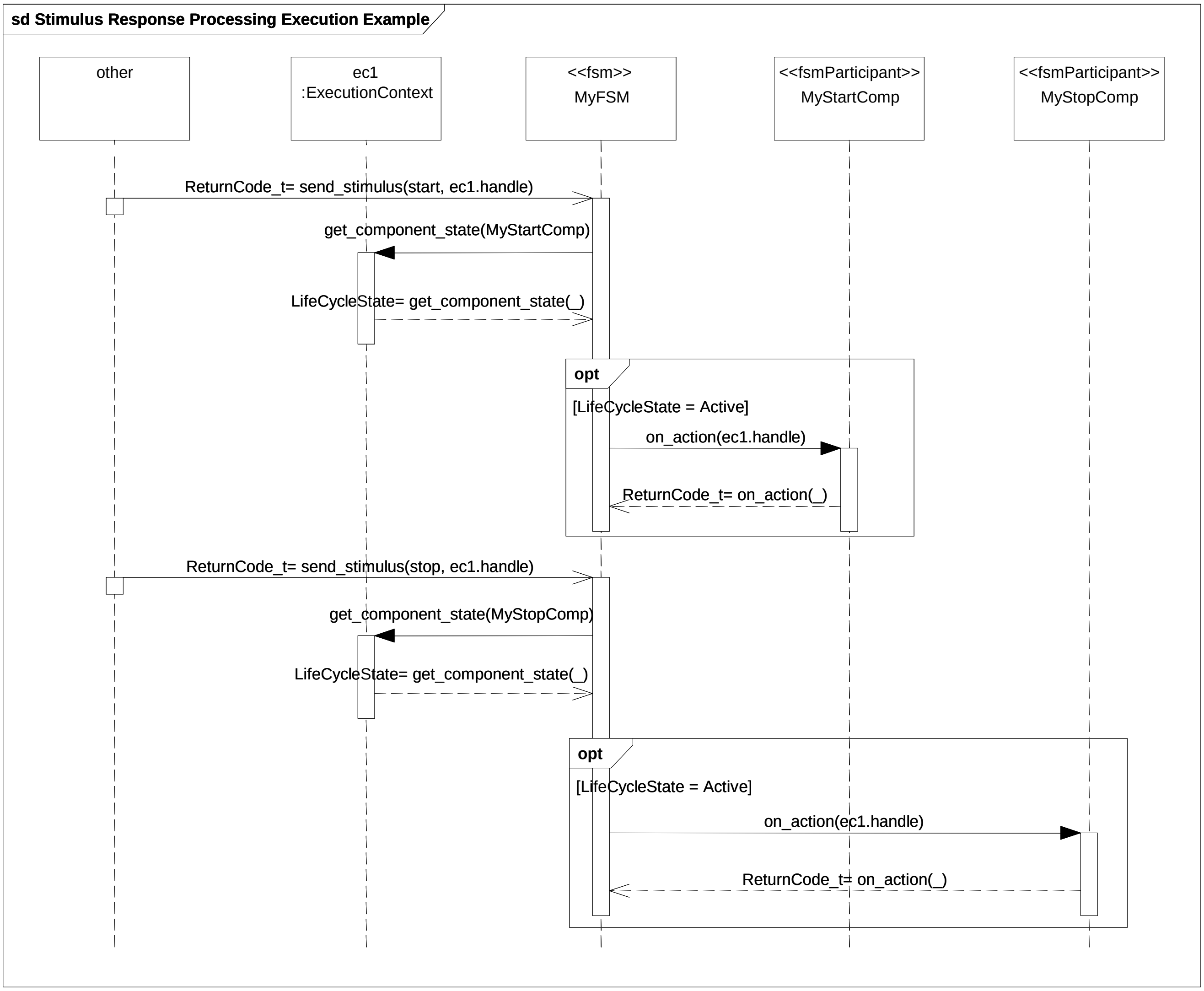

Figure 5.23 - Stimulus Responding Processing Execution Example

Constraints

- A component extended by the **`fsmParticipant`** stereotype must realize the interface **`FsmParticipantAction`**.
- A component extended by the **`fsmParticipant`** stereotype must participate in at least one execution context of kind **`EVENT_DRIVEN`**.

Generalizations

- **`lightweightRTComponent`**

5.3.2.4 FsmParticipantAction

Description

`FsmParticipantAction` is companion to **`ComponentAction`** (see Section 5.2.2.4) that is intended for use with FSM participant RTCs. It adds a callback for the interception of state transitions, state entries, and state exits.

Operations

FsmParticipantAction		
no attributes		
operations		
`on_action`		`ReturnCode_t`
	`exec_handle`	ExecutionContextHandle_t

5.3.2.4.1 on_action

Description

The indicated FSM participant RTC has been invoked as a result of a transition, state entry, or state exit in its containing FSM.

Constraints

- The given execution context shall be of kind **`EVENT_DRIVEN`**.

5.3.3 Modes of Operation

Modes of operation provide support for applications that need to switch between different implementations of a given functionality. For example, an automobile may throttle its engine either based solely on the position of the gas pedal or alternatively based on the desired speed set by the cruise control depending on whether the cruise control is activated or not. “Cruise control on” and “cruise control off” are examples of modes. (In this example, the choice of mode is a binary one, although in general any number of alternatives may be necessary.)

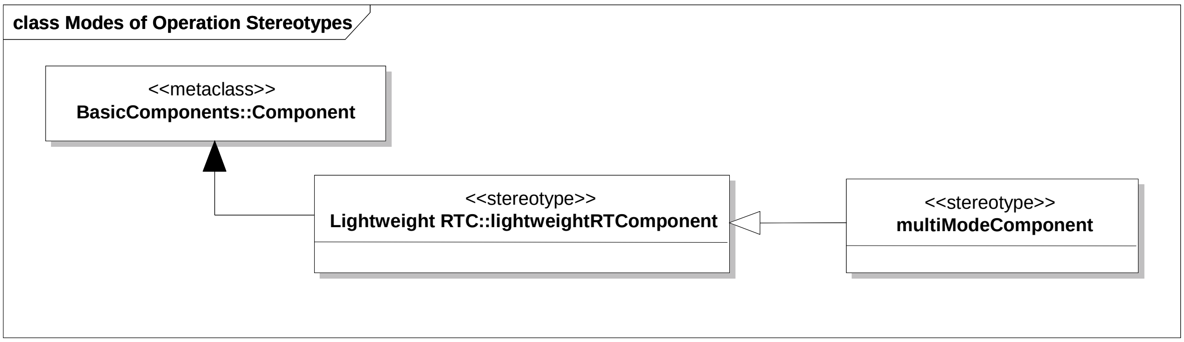

Figure 5.24 - Mode of Operation Stereotypes

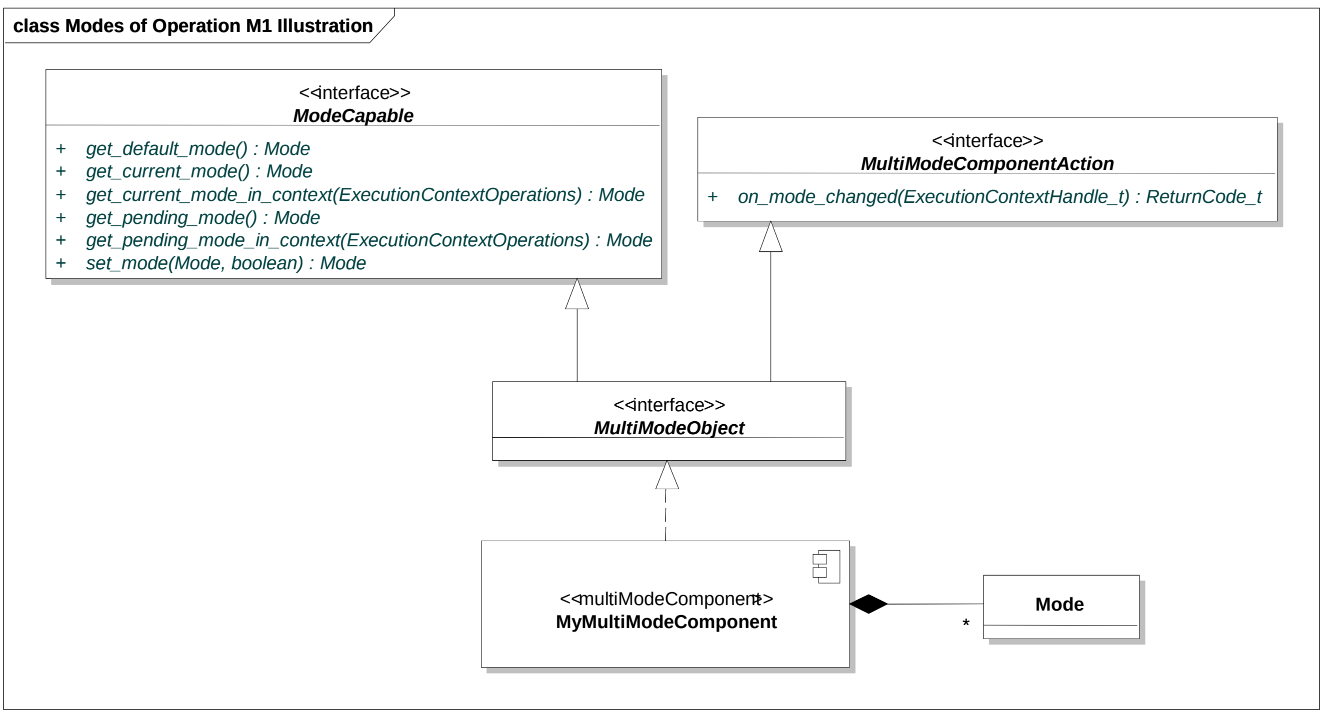

Figure 5.25 - Mode of Operation M1 Illustration

5.3.3.1 multiModeComponent

Description

An RTC developer may declare an RTC to support multiple modes by extending it with the stereotype **`multiModeComponent`**, which introduces the operations defined in the interface **`MultiModeObject`** and its supertypes.

Generalizations

- **`lightweightRTComponent`**

Constraints

- A component extended by the **`multiModeComponent`** stereotype must realize the **`MultiModeObject`** interface.

5.3.3.2 MultiModeObject

Description

The **`MultiModeObject`** interface and its supertypes define the operations that must be provided by any RTC that supports multiple modes.

Generalizations

- **`ModeCapable`**
- **`MultiModeComponentAction`**

5.3.3.3 ModeCapable

Description

The **`ModeCapable`** interface provides access to an object's modes and a means to set the current mode.

Semantics

A given RTC may support multiple modes as well as multiple execution contexts. In such a case, a request for a mode change (e.g., from "cruise control on" to "cruise control off") may come asynchronously with respect to one or more of those execution contexts. The mode of an RTC may therefore be observed to be different from one execution context to another.

- A mode is pending in a given execution context when a mode change has been requested but the new mode has not yet been observed by that context.
- The new mode has been committed in a given execution context when the context finally observes the new mode.
- The new mode has stabilized once it has been committed in all execution contexts in which the RTC participates.

Figure 5.26 depicts a state machine that describes mode changes. Each parallel region in the composite state Mode Pending represents an execution context. The trigger "sample" within that state is considered to have occurred:

- …just before the next call to **`on_execute`** (see Section 5.3.1.2.1) in the case where **`immediate`** is **`false`** and the execution kind is **`PERIODIC`**, …
- …just before the processing of the next stimulus in the case where **`immediate`** is **`false`** and the execution kind is **`EVENT_DRIVEN`**, or …
- …immediately in all other cases.

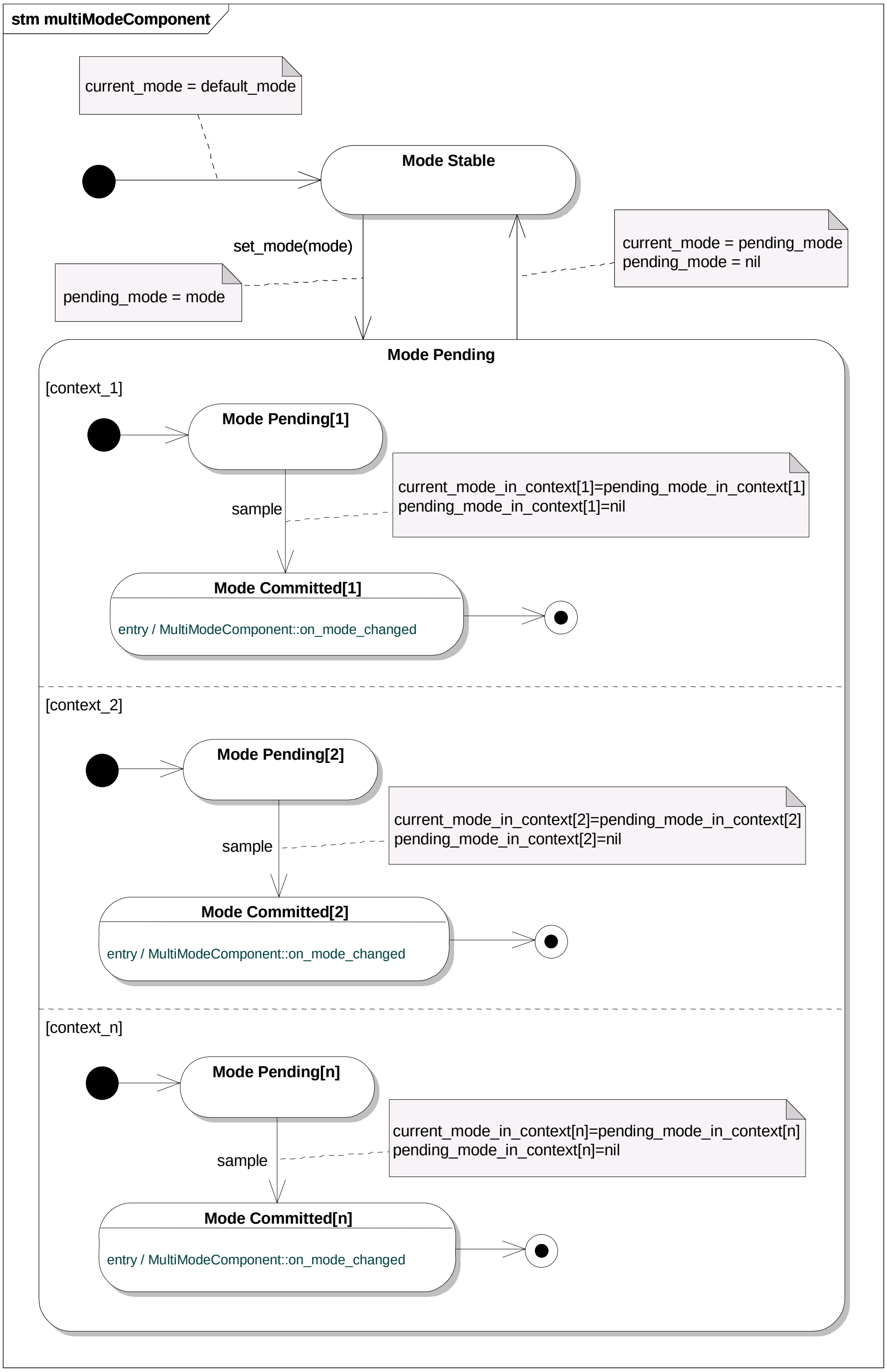

Figure 5.26 - multiModeComponent State Machine

The following figure shows a mode change on a component that is multi-mode and also a data flow participant. The mode change is not immediate, so it waits for the next "tick" of the execution context. This is a non-normative example.

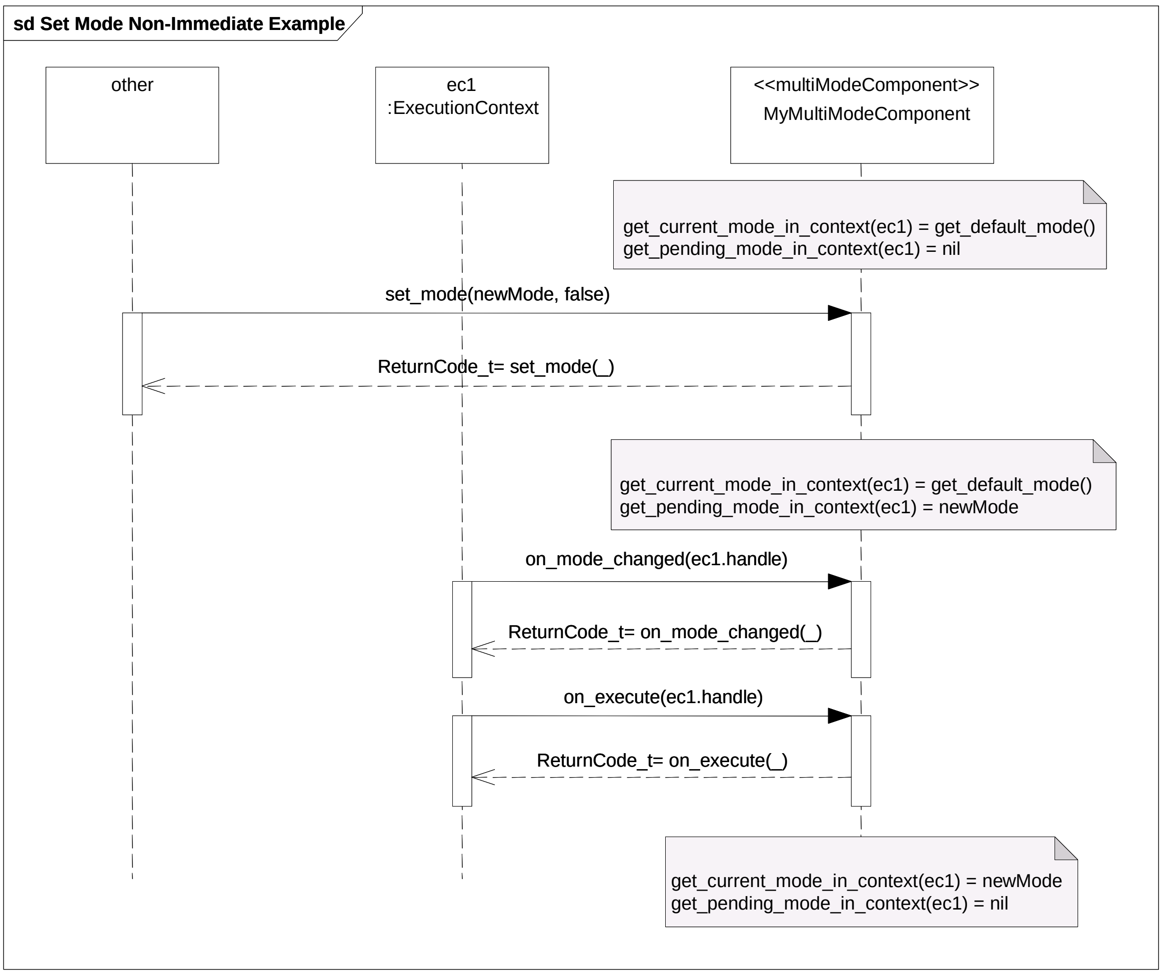

Figure 5.27 - Set Mode Non-Immediate Example

The following figure shows the same mode change, but this time it happens immediately instead of waiting for the execution context.

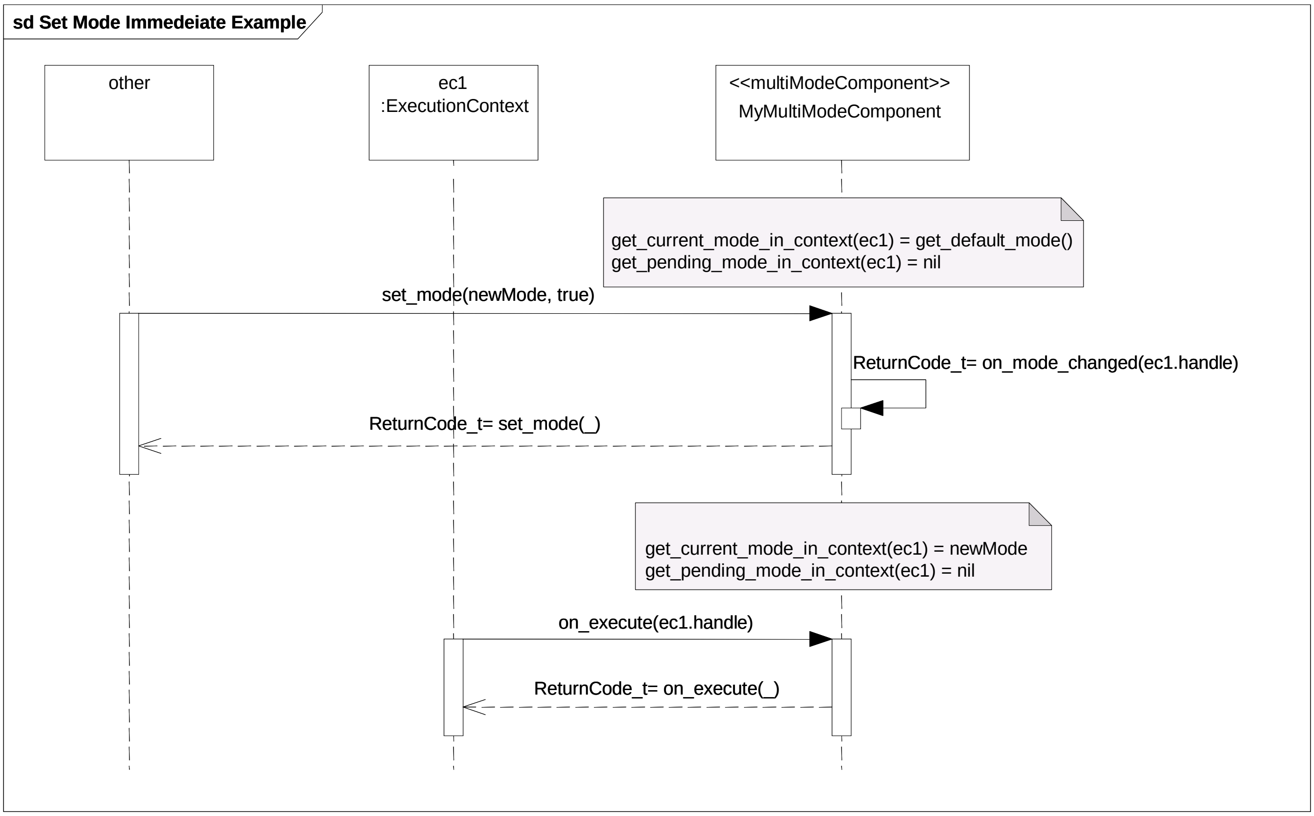

Figure 5.28 - Set Mode Immediate Example

Operations

ModeCapable		
no attributes		
operations		
get_default_mode		Mode
get_current_mode		Mode
get_current_mode_in_context		Mode
	context	ExecutionContextOperations
get_pending_mode		Mode
get_pending_mode_in_context		Mode
	context	ExecutionContextOperations
set_mode		ReturnCode_t
	mode	Mode
	immediate	Boolean

5.3.3.3.1 get_default_mode

Description

This operation shall return the mode in which the RTC shall be when no other mode has been set.

Constraints

- This operation shall not return nil.

5.3.3.3.2 get_current_mode

Description

This operation shall return the last mode to have stabilized. If no mode has been explicitly set, the current mode shall be the default mode.

Constraints

- This operation shall never return nil.

5.3.3.3.3 get_current_mode_in_context

Description

This operation returns the current mode of the component as seen by the indicated execution context.

Semantics

The manner in which this property changes is described in Figure 5.26.

5.3.3.3.4 get_pending_mode

Description

This operation shall return the last mode to have been passed to **`set_mode`** that has not yet stabilized. Once the RTC's mode has stabilized, this operation shall return nil.

5.3.3.3.5 get_pending_mode_in_context

Description

If the last mode to be requested by a call to **`set_mode`** is different than the current mode as seen by the indicated execution context (see **`get_current_mode_in_context`**), this operation returns the former. If the requested mode has already been seen in that context, it returns nil.

Semantics

See Figure 5.26 for a description of how the pending mode relates to the current mode within a given execution context.

5.3.3.3.6 set_mode

Description

This operation shall request that the RTC change to the indicated mode.

Semantics

Usually, the new mode will be *pending* in each execution context in which the component executes until the next sample period (if the execution kind is PERIODIC); at that point it will become the *current* mode in that context and there will no longer be a pending mode. However, in some cases it is important for a mode change to take place immediately; for example, a serious fault has occurred and the component must enter an emergency mode to ensure fail-safe behavior in a safety-critical system. In such a case, **`immediate`** should be **`true`** and the mode change will take place in all contexts without waiting for the next sample period.

5.3.3.4 MultiModeComponentAction

`MultiModeComponentAction` is a companion to **`ComponentAction`** that is realized by RTCs that support multiple modes.

`MultiModeComponentAction`		
no attributes		
operations		
`on_mode_changed`		`ReturnCode_t`
	`exec_handle`	ExecutionContextHandle_t

5.3.3.4.1 on_mode_changed

Description

This callback is invoked each time the observed mode of a component has changed with respect to a particular execution context.

Semantics

If the context is **`PERIODIC`**, this callback shall come before the next call to **`on_execute`** (see Section 5.3.1.2.1) within that context.

The new mode can be retrieved with **`get_current_mode_in_context`**. If the result is the same as the result of **`get_current_mode`**, the mode has stabilized.

5.3.3.5 Mode

Description

Each mode defined by a given RTC shall be represented by an instance of **`Mode`**.

5.4 Introspection

This section of the PIM extends the Lightweight RTC model and Super Distributed Objects [SDO]. It describes a capability for querying and administering RTCs at runtime. These capabilities may be used by other RTCs—in implementations that support dynamic RTC composition—but also by tools or other supporting technologies.

Introspection functionalities and “lightweight” functionalities are described separately in this specification. This design has advantages, especially for applications targeting resource-constrained environments:

- The introspection interfaces may be remote (in implementations that support that capability) without requiring that every component and supporting object be available remotely.
- Lightweight RTCs can have a much lighter footprint and can dedicate themselves more completely to their business function rather than to ancillary functionality, introspection being just one example.
- This section is divided into two subsections:
    - Resource Data Model
    - Stereotype and Interfaces

The former specifies data-only classes called “profiles” that act as descriptors for the important entities described in this document. The latter describes the behavioral classes that provide those profiles.

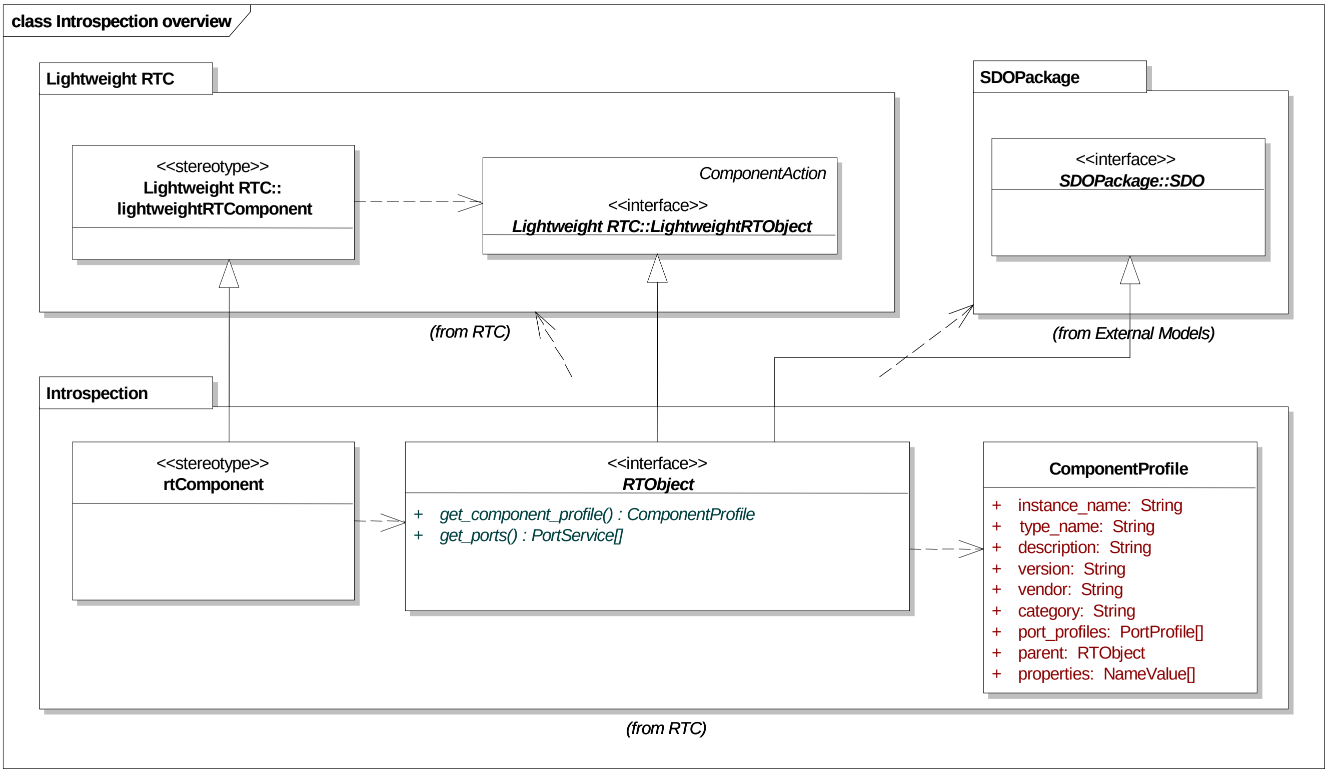

Figure 5.29 - Introspection overview

5.4.1 Resource Data Model

This section specifies the RTC resource data model, a set of data-only classes used to describe the capabilities and properties of RTCs. The RTC resource data model consists of data structure known as “profiles” that act as descriptors for the significant entities defined by the earlier portions of this specification. The data they contain may be “live” or may be populated using design-time information.

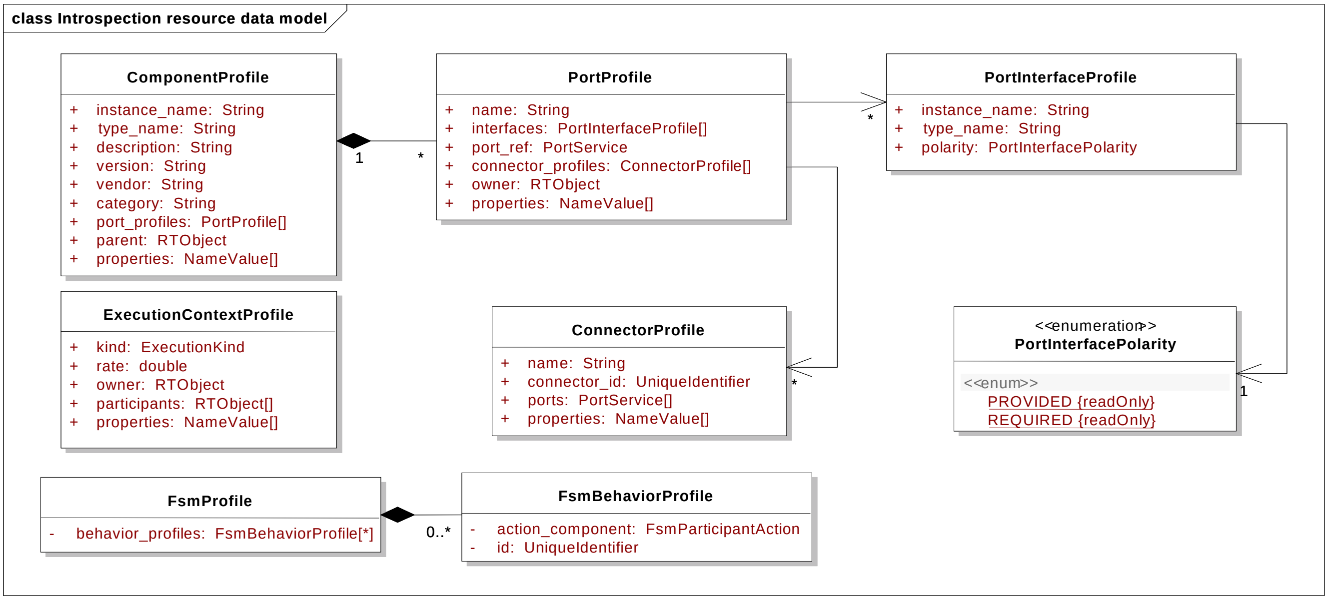

Figure 5.30 - Introspection Resource Data Model

5.4.1.1 ComponentProfile

Description

`ComponentProfile` represents the static state of an RTC that is referred to here as the “target” RTC.

Attributes

ComponentProfile	
attributes	
instance_name	String
type_name	String
description	String
version	String
vendor	String
category	String

port_profiles	PortProfile[]
parent	RTObject
properties	NameValue[]
no operations	

5.4.1.1.1 instance_name

Description

This attribute shall contain the name of the target RTC instance.

Semantics

The **instance_name** should be unique among RTC instances contained within the same containing component.

5.4.1.1.2 type_name

Description

This attribute shall contain the name of the target RTC class.

Semantics

Each RTC class must have a name that is unique within an application.

5.4.1.1.3 description

Description

This attribute shall briefly describe the target RTC for the benefit of a human operator.

5.4.1.1.4 version

Description

This attribute shall contain the version number of the target RTC class.

Semantics

The format of the version number is outside of the scope of this specification.

5.4.1.1.5 vendor

Description

The name of the individual or organization that produced the target RTC class.

5.4.1.1.6 category

Description

This attribute contains the name of a “category” or group to which the target RTC belongs.

5.4.1.1.7 port_profiles

Description

This attribute contains a list of `PortProfile`s that describe the ports of the target RTC.

Semantics

There shall be a one-to-one correspondence between the members of this list and the ports of the target RTC.

5.4.1.1.8 parent

Description

This attribute contains a reference to the RTC that contains the target RTC instance. If the target RTC instance is not owned by any other RTC, this field stores a nil reference.

5.4.1.1.9 properties

Description

This attribute contains additional properties of the target RTC.

Semantics

This attribute provides implementations the opportunity to describe additional characteristics of a particular RTC that are otherwise outside of the scope of this specification.

5.4.1.2 PortProfile

Description

A PortProfile describes a port of an RTC (referred to as the “target” RTC). This port is referred to as the “target” port.

From this profile, other components and tools can obtain Port’s name, type, object reference, and so on.

Attributes

PortProfile	
attributes	
name	String
interfaces	PortInterfaceProfile[]
port_ref	PortService
connector_profiles	ConnectorProfile[]
owner	RTObject
properties	NameValue[]
no operations	

5.4.1.2.1 name

Description

This attribute contains the name of the target port.

Semantics

Ports owned by an RTC are distinguished by their names. Therefore, this name should be unique within the target RTC.

5.4.1.2.2 interfaces

Description

This attribute contains the name and polarity of each interface exposed by the target port.

5.4.1.2.3 port_ref

Description

This attributes contains a reference to the target port.

5.4.1.2.4 connector_profiles

Description

This attribute contains a collection of profiles describing the connections to the target port.

5.4.1.2.5 owner

Description

This attribute contains a reference to the target RTC.

5.4.1.2.6 properties

Description

This attribute contains additional properties of the port.

Semantics

This attribute provides implementations the opportunity to describe additional characteristics of a particular port that are otherwise outside of the scope of this specification.

5.4.1.3 PortInterfaceProfile

Description

`PortInterfaceProfile` describes an instance of a particular interface as it is exposed by a particular port. These objects are referred to below as the "target interface" and "target port" respectively.

Attributes

PortInterfaceProfile	
attributes	
instance_name	String
type_name	String
polarity	PortInterfacePolarity
no operations	

5.4.1.3.1 instance_name

Description

This attribute stores the name of the target interface instance.

5.4.1.3.2 type_name

Description

This attribute stores the name of the target interface type.

5.4.1.3.3 polarity

Description

This attribute indicates whether the target interface instance is provided or required by the RTC.

5.4.1.4 PortInterfacePolarity

Description

The **`PortInterfacePolarity`** enumeration identifies exposed interface instances as provided or required.

Attributes

PortInterfacePolarity	
attributes	
PROVIDED	PortInterfacePolarity
REQUIRED	PortInterfacePolarity
no operations	

5.4.1.4.1 PROVIDED

Description

The target interface is provided as an output by the target port.

5.4.1.4.2 REQUIRED

Description

The target interface is required as an input by the target port.

5.4.1.5 ConnectorProfile

Description

The **`ConnectorProfile`** contains information about a connection between the ports of collaborating RTCs.

Attributes

ConnectorProfile	
attributes	
name	String
connector_id	UniqueIdentifier
ports	PortService[]
properties	NameValue[]
no operations	

5.4.1.5.1 name

Description

This attribute contains the name of this connection.

5.4.1.5.2 connector_id

Description

Each connector has a unique identifier that is assigned when connection is established. This attribute stores that identifier.

5.4.1.5.3 ports

Description

This field stores references to all ports connected by the target connector.

5.4.1.5.4 properties

Description

This attribute contains additional properties of the connection.

Semantics

This attribute provides implementations the opportunity to describe additional characteristics of a particular connection that are outside of the scope of this specification.

5.4.1.6 ExecutionContextProfile

Attributes

ExecutionContextProfile
attributes

kind	ExecutionKind
rate	Double
owner	RTObject
participants	RTObject[]
properties	NameValue[]
no operations	

5.4.1.6.1 kind

Description

This attribute stores the context's **ExecutionKind**.

5.4.1.6.2 rate

Description

This attribute stores execution rate.

Semantics

If the execution kind is not **PERIODIC**, the value here may not be valid (and should be negative in that case). See **ExecutionContext::get_rate** (see Section 5.2.2.6.4) and **set_rate** (see Section 5.2.2.6.5) for more information.

5.4.1.6.3 owner

Description

This attribute stores a reference to the RTC that owns the context.

5.4.1.6.4 participants

Description

This attribute stores references to the context's participant RTCs.

5.4.1.6.5 properties

Description

This attribute contains additional properties of the execution context.

Semantics

This attribute provides implementations the opportunity to describe additional characteristics of a particular execution context that are outside the scope of this specification.

5.4.1.7 FsmProfile

Description

The **`FsmProfile`** describes the correspondence between an FSM and its contained FSM participants. This Profile is necessary for Stimulus Response Processing.

Attributes

FsmProfile	
attributes	
behavior_profiles	FsmBehaviorProfile[]
no operations	

5.4.1.7.1 behavior_profiles

Description

This attribute lists the correspondences between an FSM and its contained FSM participants.

5.4.1.8 FsmBehaviorProfile

Description

`FsmBehaviorProfile` represents the association of an FSM participant with a transition, state entry, or state exit in an FSM.

Semantics

The assignment of identifiers to particular transitions, state entries, or state exits is implementation-dependent.

Attributes

FsmBehaviorProfile	
attributes	
action_component	FsmParticipantAction
id	UniqueIdentifier
no operations	

5.4.1.8.1 action_component

Description

This attribute stores a reference to the FSM participant that is invoked when the containing Fsm receives a message distinguished by id.

5.4.1.8.2 id

Description

This attribute stores the message identifier.

5.4.2 Stereotypes and Interfaces

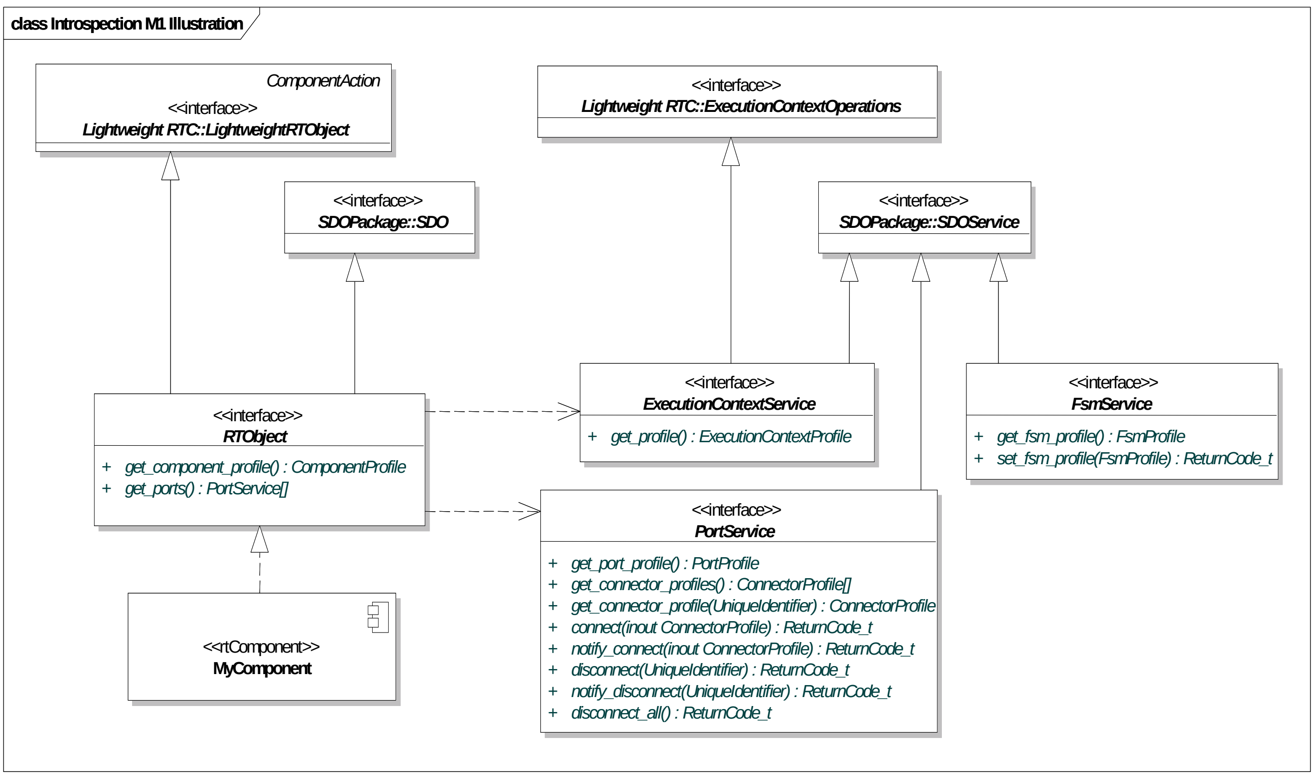

Figure 5.31 - Introspection M1 Illustration

Figure 5.32 - Introspection Stereotypes

5.4.2.1 rtComponent

Description

The **rtComponent** stereotype identifies the component it extends as an RTC that realizes an SDO-based introspective interface.

Generalizations

- **lightweightRTComponent**

Constraints

- A component extended by the **rtComponent** stereotype must realize the **RTObject** interface.

5.4.2.2 RTObject

Description

The **RTObject** interface defines the operations that all SDO-based RTCs must provide. It is required by the **rtComponent** stereotype.

Generalizations

- **LightweightRTObject**
- **SDO** [SDO]

Constraints

- Any execution contexts returned from the inherited methods **LightweightRTObject::get_owned_contexts** and **LightweightRTObject::get_participating_contexts** shall be of type **ExecutionContextService**.

Operations

RTObject		
no attributes		
operations		
get_component_profile		ComponentProfile
get_ports		PortService[]

5.4.2.2.1 get_component_profile

Description

This operation returns the **ComponentProfile** of the RTC.

5.4.2.2.2 get_ports

Description

This operation returns a list of the RTCs ports.

5.4.2.3 PortService

Description

An instance of the **PortService** interface represents a port (i.e., **UML::Composite Structures::Ports::Port**) of an RTC. It provides operations that allow it to be connected to and disconnected from other ports.

Semantics

A port service can support unidirectional or bidirectional communication.

A port service may allow for a service-oriented connection, in which other connected ports, invoke methods on it. It may also allow for a data-centric connection, in which data values are streamed in or out. In either case, the connection is described by an instance of **ConnectorProfile**. However, the behavioral contracts of such connections are dependent on the interfaces exposed by the ports and are not described normatively by this specification.

Generalizations

- **SDOService** [SDO]

Operations

PortService
no attributes
operations

get_port_profile		PortProfile
get_connector_profiles		ConnectorProfile[]
get_connector_profile		ConnectorProfile
	connector_id	UniqueIdentifier
connect		ReturnCode_t
	connector_profile	ConnectorProfile
notify_connect		ReturnCode_t
	inout connector_profile	ConnectorProfile
disconnect		ReturnCode_t
	connector_id	UniqueIdentifier
notify_disconnect		ReturnCode_t
	connector_id	UniqueIdentifier
disconnect_all		ReturnCode_t

5.4.2.3.1 get_port_profile

Description

This operation returns the **PortProfile** of the **PortService**.

5.4.2.3.2 get_connector_profiles

Description

This operation returns a list of the **ConnectorProfile**s of the **PortService**.

5.4.2.3.3 get_connector_profile

Description

This operation returns the **ConnectorProfile** specified by a connector ID.

5.4.2.3.4 connect

Description

This operation establishes connection between this port and the peer ports according to given **ConnectionProfile**.

Semantics

A **ConnectorProfile** has a sequence of port references. This port invokes the **notify_connect** operation of one of the ports included in the sequence. It follows that the notification of connection is propagated by the **notify_connect** operation with **ConnectorProfile**. This operation returns **ConnectorProfile** return value and returns **ReturnCode_t** as return codes.

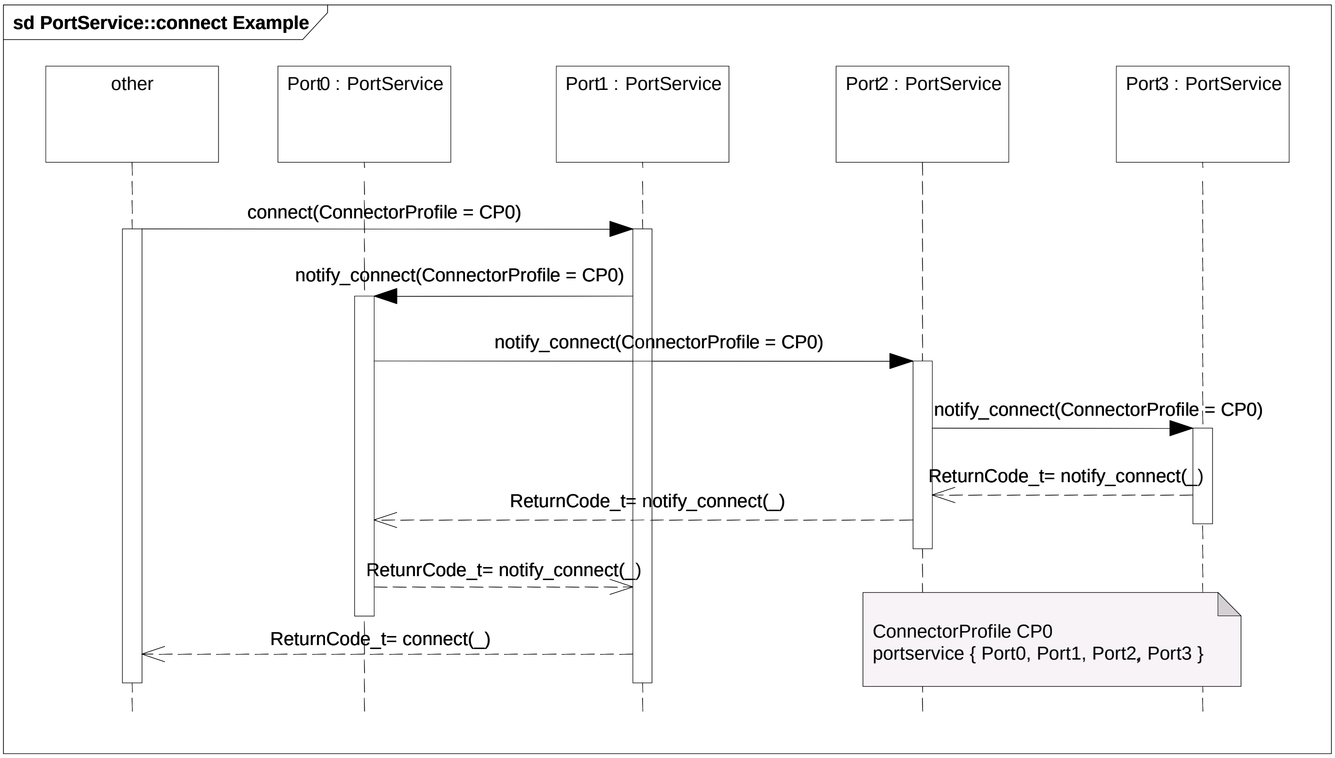

Figure 5.33 - PortService::connect Example

5.4.2.3.5 disconnect

Description

This operation destroys the connection between this port and its peer ports using the ID that was given when the connection was established.

Semantics

This port invokes the **`notify_disconnect`** operation of one of the ports included in the sequence of the **`ConnectorProfile`** stored when the connection was established. The notification of disconnection is propagated by the **`notify_disconnect`** operation.

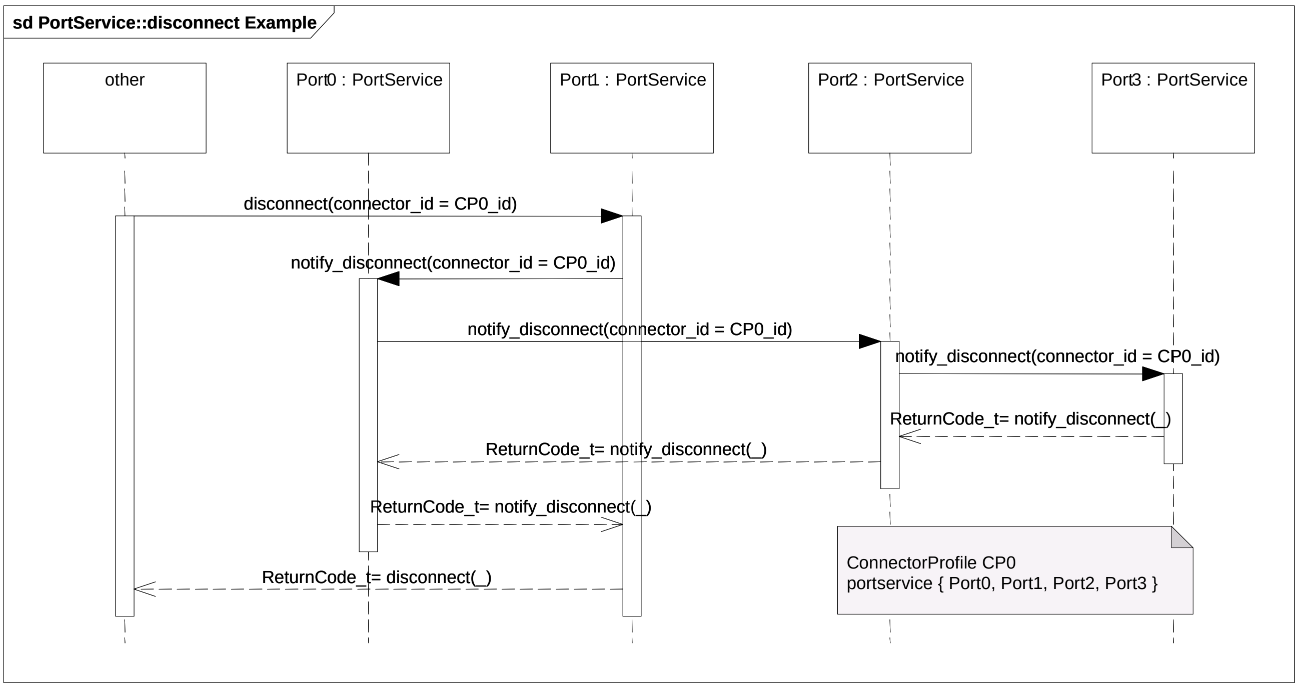

Figure 5.34 - PortService::disconnect Example

5.4.2.3.6 disconnect_all

Description

This operation destroys all connection channels owned by the **`PortService`**.

5.4.2.3.7 notify_connect

Description

This operation notifies this **`PortService`** of the connection between its corresponding port and the other ports and propagates the given **`ConnectionProfile`**.

Semantics

A ConnectorProfile has a sequence of port references. This **`PortService`** stores the **`ConnectorProfile`** and invokes the **`notify_connect`** operation of the next **`PortService`** in the sequence. As ports are added to the connector, **`PortService`** references are added to the **`ConnectorProfile`** and provided to the caller. In this way, notification of connection is propagated with the **`ConnectorProfile`**.

5.4.2.3.8 notify_disconnect

Description

This operation notifies a **`PortService`** of a disconnection between its corresponding port and the other ports. The disconnected connector is identified by the given ID, which was given when the connection was established.

Semantics

This port invokes the **notify_disconnect** operation of the next **PortService** in the sequence of the **ConnectorProfile** that was stored when the connection was established. As ports are disconnected, **PortService** references are removed from the **ConnectorProfile**. In this way, the notification of disconnection is propagated by the **notify_disconnect** operation.

5.4.2.4 ExecutionContextService

Description

An **ExecutionContextService** exposes an **ExecutionContext** as an SDO service such that the context may be controlled remotely.

Semantics

Depending on the implementation, this interface may itself be an execution context (that is, it may be passed to the operations of **ComponentAction**) or it may represent a remote execution context that is not of type **ExecutionContextService**.

Interface Realizations

- **ExecutionContextOperations**

Generalizations

- **SDOService** [SDO]

Operations

ExecutionContextService		
no attributes		
operations		
get_profile		ExecutionContextProfile

5.4.2.4.1 get_profile

Description

This operation provides a profile "descriptor" for the execution context.

5.4.2.5 FsmService

Description

The **FsmService** interface defines operations necessary for Stimulus Response Processing as an SDO service.

Generalizations

- **`SDOService`** [SDO]

Operations

FsmService		
no attributes		
operations		
`get_fsm_profile`		`FsmProfile`
`set_fsm_profile`		`ReturnCode_t`
	`fsm_profile`	`FsmProfile`

5.4.2.5.1 get_fsm_profile

Description

Get the current state of the FSM.

Semantics

Modifications to the object returned by this operation will not be reflected in the FSM until and unless **`set_fsm_profile`** is called.

5.4.2.5.2 set_fsm_profile

Description

This operation will be used to modify the behavior of an FSM as described in Stimulus Response Processing.

6 Platform Specific Models

In order to maximize interoperability, this document describes three PSMs that should be considered normative in sections 6.3, 6.4, and 6.5. They correspond to the PSM conformance points outlined in Chapter 2. All PSMs draw on a common set of IDL definitions, which is presented first.

6.1 UML-to-IDL Transformation

The PSMs below require IDL definitions for the interfaces, data types, and other model elements from the PIM. They also require IDL representations of the model elements from [UML] on which the PIM depends: Component, Port, Connector, etc. Representing all of the UML in IDL is beyond the scope of this specification. This specification takes a more parsimonious approach.

- IDL definitions for the elements from this specification are provided explicitly in Section 6.2.
- Mapping rules from a subset of UML to IDL are provided in this section. Only those parts of UML that are necessary to describe the PIM or critical for the definition of conforming components are described here. Mappings of all other UML constructs are implementation-defined.

6.1.1 Basic Types and Literals

Primitive types (see Section 5.2.3) shall map to the corresponding IDL primitive types. Specifically:

- Character: **char**
- Double: **double**
- Float: **float**
- Long: **long**
- LongDouble: **long double**
- LongLong: **long long**
- Octet: **octet**
- Short: **short**
- UnsignedLong: **unsigned long**
- UnsignedLongLong: **unsigned long long**
- UnsignedShort: **unsigned short**
- WideCharacter: **wchar**
- WideString: **wstring**

The standard UML String and Boolean types are also used by this specification.

- Boolean: **bool**
- String: **string**

The literal specifications defined in the PIM (see Section 5.2.4)—as well as those referenced from [UML]—shall be represented as IDL literal values.

6.1.2 Classes and Interfaces

UML classes and interfaces shall be represented as IDL interfaces of the same name.

- Each operation or attribute on the UML classifier shall be represented by a corresponding operation or attribute in IDL.
- The general classifiers of UML classes and interfaces shall be represented as inheritance between the corresponding IDL interfaces.

6.1.3 Components

RT components shall be represented as IDL components of the same name. As required by the PIM, this component must support the **LightweightRTObject** interface or some subtype thereof, such as **RTObject**.

6.1.3.1 Component Inheritance

For each of the UML component's general classifiers that can be represented as a UML interface according to the mapping rules here, the corresponding IDL component shall support the corresponding IDL interface.

If the RTC has a general classifier that is another RTC, the IDL component corresponding to the former shall inherit from that corresponding to the latter.

6.1.3.2 Component Ports

Ports providing a single interface shall be represented as facets. Ports requiring a single interface shall be represented as receptacles. Receptacles shall be multiplex unless otherwise constrained by the model. These facets and receptacles shall have the same names as the UML ports.

Ports that expose multiple interfaces, possibly of different polarities, cannot currently be represented in IDL. Therefore, such ports shall be divided into separate facets and receptacles, one per exposed interface, using the following name transformation:

- The name of the port,
- An underscore,
- The name of the interface's association end.

For example:

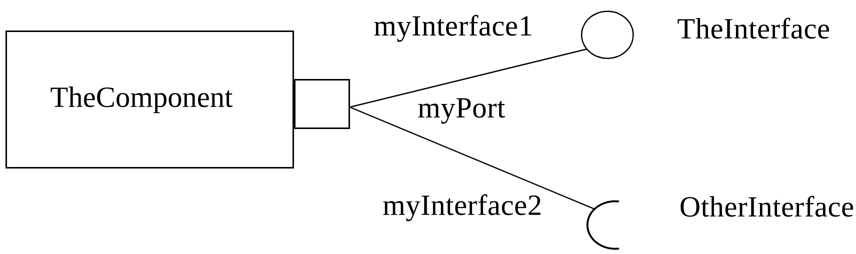

Figure 6.1 - Example of port name mapping

The component in Figure 6.1 would be represented in IDL as follows:

```
// interfaces TheInterface and OtherInterface previously defined

component TheComponent {
    provides TheInterface myPort_myInterface1;
    uses multiple OtherInterface myPort_myInterface2;
};
```

6.1.3.3 Composite Components

Implementations may choose to represent composite components—those whose internal parts are interconnected instances of other components—using declarative descriptions of component assemblies (as in [DC]), generated code, or some other means. Deployment and configuration of RTCs is beyond the scope of this specification.

6.1.4 Enumerations

Enumerations in the PIM shall be represented as IDL enumerations of the same name. Each attribute shall correspond to a constant within that enumeration.

6.1.5 Packages

A UML package shall be represented by an IDL module of the same name.

6.2 IDL Definitions

This section defines the representations of the PIM elements in IDL. These definitions have been created using the rules from Section 6.1 as extended below. Several minor changes have been made from the PIM in order to make the appearance of the API more natural in IDL (and in programming language APIs generated from it).

The IDL definitions themselves may be found in an annex to this document as well as in the accompanying file RTC.idl (see *http://www.omg.org/spec/RTC/20070801*). Only the latter is normative.

6.2.1 Classes and Interfaces

Because UML classes and interfaces are both represented by IDL interfaces (see Section 6.1.2), the types **`ExecutionContext`** and **`ExecutionContextOperations`** are essentially identical. Therefore, in the PSMs, they have been collapsed to a single IDL interface "**`ExecutionContext`**."

6.2.2 Stereotypes

Stereotypes have been represented as IDL interfaces intended to be supported by components having those stereotypes. This convention provides a convenient way to add the necessary operations to supporting components. Two additional rules apply:

- Stereotypes that constrain their extended types to realize a given interface have been unified with that interface.
- IDL interfaces representing UML stereotypes are named with an initial capital letter to reflect common naming conventions for IDL interfaces.

Specifically:

- **`lightweightRTComponent`** has been unified with the **`LightweightRTObject`** interface.
- **`dataFlowComponent`** is represented by the **`DataFlowComponent`** interface.
- **`fsm`** is represented by the **`Fsm`** interface.
- **`fsmParticipant`** is represented by the **`FsmParticipant`** interface.
- **`multiModeComponent`** has been unified with the **`MultiModeComponentObject`** interface.
- **`rtComponent`** has been unified with the **`RTObject`** interface.

6.2.3 Return Codes

The PIM allows **`ReturnCode_t`** to be modeled in a PSM as either an actual return value or as an exception. The IDL definition chooses a return value as the lowest common denominator mapping across programming languages, and therefore retains the enumeration representation from Section 6.1.4. However, in an annex and RTC.idl, since a word confliction is expected in mapping to other languages, **`OK`** and **`ERROR`** are replaced to **`RTC_OK`** and **`RTC_ERROR`**. This convention should not be construed to overrule the more general allowance in the PIM.

6.2.4 Packages

In the PIM, the RTC package has sub-packages. These sub-packages have been flattened in the PSMs in order to simplify the API in those language bindings (such as C) that reflect IDL modules in contained identifier names.

6.3 Local PSM

RTCs implemented using this PSM communicate within a single local application without intervening network communication; RTCs, assembly connectors, and delegation connectors are implemented as local objects. Specifically, the presence of a CORBA ORB shall not be required.

This communication model is especially important for RTCs with very strict timeliness, reliability, and determinism requirements. It is also applicable to resource-constrained environments. (The default communication styles in Simulink from The MathWorks, LabView from National Instruments, and Constellation from RTI are analogous to this PSM.)

6.3.1 IDL Transformation Rules

The programming interfaces for this PSM are implied by the IDL and IDL mapping rules from Section 6.1 and Section 6.2 as extended below. One additional exception applies: because this PSM is not CORBA-based, any CORBA-specific base types supplied by the IDL-to-programming language mapping shall be omitted from any generated code.

This PSM relies on IDL type definitions to provide a programming language-agnostic programming interface to an RTC-based middleware. Therefore, it is only well-defined in the context of a particular IDL-to-programming language mapping. To derive the RTC API in the target programming language, that mapping shall be applied to the input IDL with the additional restrictions noted in this section. These restrictions stem from the need to eliminate usages of CORBA-specific types that may be incidentally implied by a particular IDL-to-programming language mapping.

The restrictions in this section currently apply to the C++ language only, and specifically to the OMG-standard IDL-to-C++ mapping, although future revisions may introduce explicit support for additional programming language mappings. Until that time, conforming implementations of this PSM with other programming language mappings will not be possible.

6.3.1.1 Mapping for Interfaces and Reference Types

6.3.1.1.1 Implicit Local Interfaces

Since object distribution is beyond the scope of this PSM, all IDL interfaces shall be treated as local interfaces with respect to code generation. That is, any code generated shall be as if the interface declaration was prefixed with the IDL local keyword.

6.3.1.1.2 CORBA-Specific Base Interfaces

The class generated for an IDL interface shall not extend any CORBA-specific base class such as **`CORBA::Object`** or **`CORBA::LocalObject`**. The programming interfaces provided by those base classes are therefore non-normative with respect to this PSM.

- Nil values shall be represented by the value **`NULL`**, eliminating the need for **`is_nil`** and similar operations.
- Object references are represented as simple pointers (see Section 6.3.1.1.4), and can therefore be compared directly, eliminating the need for **`CORBA::Object::is_equivalent`**.
- Runtime type identification is beyond the scope of this PSM. Implementations that require it must either rely on the support built into the target programming language or provide an alternative implementation-specific mechanism.

6.3.1.1.3 Abstract Interfaces

This specification does not use any abstract interfaces, nor does any specification on which it depends. Furthermore, abstract interfaces do not result from any of the UML-to-IDL transformation rules in this specification. Therefore, the IDL mapping of abstract interfaces is outside the scope of this PSM.

6.3.1.1.4 Pointer Types

Support for the "smart pointer" **`_ptr`**, **`_var`**, and **`_out`** types shall not be required. Instead, object references shall be represented with native pointers. Memory management is the responsibility of the party that allocates it: either the middleware or the component. This convention applies not only to interface types, but also to strings, wide strings, and other non-basic types.

Because pointers are used directly, the **`_duplicate`** operation on interface types is not needed and is therefore not normative with respect to this PSM.

6.3.1.2 Mapping for Basic Data Types

IDL basic data types shall map to type definitions in the **`RTC`** namespace having the same names as the corresponding PIM types:

- **short**: `RTC::Short`
- **long**: `RTC::Long`
- **long long**: `RTC::LongLong`
- **unsigned short**: `RTC::UnsignedShort`
- **unsigned long**: `RTC::UnsignedLong`
- **unsigned long long**: `RTC::UnsignedLongLong`
- **float**: `RTC::Float`
- **double**: `RTC::Double`
- **long double**: `RTC::LongDouble`
- **char**: `RTC::Char`
- **wchar**: `RTC::WideChar`
- **boolean**: `RTC::Boolean`
- **octet**: `RTC::Octet`

Programmers concerned with portability should use the **`RTC`** types. However, some may feel that using these types with the **`RTC`** qualification impairs readability. As in the OMG-standard IDL-to-C++ mapping, on platforms where the C++ data type is guaranteed to be identical to the OMG IDL data type, a compliant implementation may generate the native C++ type.

6.3.1.3 Mapping for String Types

IDL strings map to **`char*`** as in the OMG-standard IDL-to-C++ mapping. For dynamic allocation of strings, compliant implementations must use the following functions from the RTC namespace:

```
// C++
namespace RTC {
    char* string_alloc(RTC::UnsignedLong len);
    char* string_dup(const char*);
```

```
    void string_free(char*);
}
```

The behavior of these functions shall be the same as the equivalent functions defined in the OMG-standard IDL-to-C++ mapping specification.

6.3.1.4 Mapping for Wide String Types

IDL wide strings, whether bounded or unbounded, map to **`RTC::WideChar*`**. For dynamic allocation of wide strings, compliant implementations must use the following functions from the **`RTC`** namespace:

```
// C++
namespace RTC {
    RTC::WideChar* wstring_alloc(RTC::UnsignedLong len);
    RTC::WideChar* wstring_dup(const RTC::WideChar*);
    void wstring_free(RTC::WideChar*);
}
```

The behavior of these functions shall be the same as the equivalent functions defined in the OMG-standard IDL-to-C++ mapping specification.

6.3.1.5 Mapping for Fixed Types

A mapping for the IDL fixed point type is outside of the scope of this PSM.

6.3.1.6 Mapping for Any Type

The IDL any type shall be represented in generated code as an instance of the type **`RTC::Any`**. The programming interfaces and semantics are as described in the OMG-standard IDL-to-C++ mapping; only the namespace is changed.

As described elsewhere, the types **`CORBA::AbstractBase`**, **`CORBA::ValueBase`**, **`CORBA::TypeCode`**, and **`CORBA::Fixed`** are non-normative with respect to this PSM. Therefore, the **`Any`** member structures and operations that refer to these types are also non-normative.

6.3.1.7 Mapping for Valuetypes

IDL **valuetype**s are used neither by this specification nor by the SDO specification. Furthermore, they do not result from any of the UML-to-IDL mappings defined for this specification. Therefore, the IDL-to-programming language mappings for **valuetype**s are beyond the scope of this PSM.

6.3.1.8 Exceptions

Operations defined in this specification report errors by means of **`ReturnCode_t`** objects, not IDL exception objects. They shall therefore refrain from throwing system exceptions and shall instead fail with the appropriate **`ReturnCode_t`** result, such as **`BAD_PARAMETER`**, **`OUT_OF_RESOURCES`**, or simply **`ERROR`**.

However, the SDO specification does rely on exceptions for error reporting. Additionally, this PSM specifies that certain programming interfaces defined by the OMG-standard IDL-to-C++ mapping (identified elsewhere in this PSM) shall be provided in the RTC namespace. Unless otherwise noted, these APIs do not use **`ReturnCode_t`** to report errors. Therefore, the types **`Exception`**, **`SystemException`**, **`CompletionStatus`**, and **`UserException`** shall be defined in the RTC namespace. These shall take the place of their CORBA namespace equivalents in all programming interfaces, including as base classes for user-defined and standard exception types. They shall function identically to their corresponding types from the CORBA namespace. Standard subclasses of `SystemException` shall also exist in the `RTC` namespace rather than in the `CORBA` namespace.

The OMG-standard IDL-to-C++ mapping allows for C++ implementations that do not support exception handling. This allowance is made through the provision of an `Environment` pseudo-interface. Implementations of this PSM that do not support exceptions shall provide a class `RTC::Exception` that is functionally equivalent to `CORBA::Environment`.

6.3.1.9 Mapping for Components

The mapping for IDL component-related concepts relies on the "equivalent interfaces" outlined in the CCM specification. These IDL-to-IDL transformations remain normative.

The class generated for an IDL component shall not extend any CORBA- or CCM-specific base class such as `Components::CCMObject`. The programming interfaces provided by that base class, and its base classes, are therefore non-normative with respect to this PSM.

6.3.1.9.1 Mapping for Receptacles

Simplex Receptacles

A uses declaration of the following form:

```
uses <interface_type> <receptacle_name>;
```

results in the following equivalent operations defined in the component interface.

```
ReturnCode_t connect_<receptacle_name>(
   in <interface_type> conxn);
<interface_type> disconnect_<receptacle_name>();
<interface_type> get_connection_<receptacle_name>();
```

These definitions match those defined by the Lightweight CCM specification except with respect to error reporting. They report error conditions using the `ReturnCode_t` mechanism defined in this specification rather than with explicit exceptions. The disconnect operation shall indicate failure with a nil return result.

Multiplex Receptacles

A uses declaration of the following form:

```
uses multiple <interface_type> <receptacle_name>;
```

results in the following equivalent operations defined in the component interface.

```
struct <receptacle_name>Connection {
   <interface_type> objref;
   RTC::Cookie ck;
};

sequence<<receptacle_name>Connection> <receptacle_name>Connections;
```

```
RTC::Cookie connect_<receptacle_name>(
    in <interface_type> connection);
<interface_type> disconnect_<receptacle_name>(
    in RTC::Cookie ck);
<receptacle_name>Connections get_connections_<receptacle_name>();
```

The above declarations are similar to those defined by the Lightweight CCM specification with two exceptions: error handling and the replacement of `Components::Cookie` with `RTC::Cookie`. The latter shall be identical to the former apart from the change in namespace.

The `disconnect` operation shall report errors by means of a nil return result.

6.3.2 Interfaces

Existing modeling and development tools used by robotics developers, such as Simulink from The MathWorks and LabView from National Instruments, often rely on data-only connections between program modules. The connection types in such cases are not interfaces in the IDL sense; they may be primitives, arrays, or simple structures. This approach has several benefits from an implementation standpoint:

- Robotics developers, who are often not software engineers, think in data-centric terms, so this ability is familiar and comfortable for them.
- Using unencapsulated data decreases the number of function calls necessary along the critical path of a high-rate real-time application.
- Legacy robotics code is frequently not interface-centric, and integrating with it is simpler if legacy types can be used directly.

For these reasons, UML interfaces may be represented by any data type that can be expressed in the target language as long as it fulfills the contract described by the interface.

6.3.3 Connectors

Connectors shall be represented as instances of the type of the connected ports. This type may contain data, methods, or both. If the type of the connector contains data, implementations must ensure that all ports connected to it observe the same values at all times.

If the type of the connector contains methods, the connector implementation is responsible for performing any multiplexing or de-multiplexing of method calls that may be necessary when a connector is not one-to-one. When a single caller (i.e., an operation defined by a required interface) is connected to multiple callees (i.e., the same operation within multiple instances of the matching provided interface), the implementation must invoke all callee methods. However, the handling of their return results or exceptions is implementation-defined.

6.3.4 SDO

The Introspection package (Section 5.4) of the PIM relies on the Super Distributed Objects specification [SDO]. Implementations that support that package shall use the IDL provided with that specification's CORBA PSM as transformed according to this PSM.

6.4 Lightweight CCM PSM

In this PSM, RTCs are mapped to CCM components supporting the relevant IDL interfaces described in Section 6.1 and Section 6.2.

6.4.1 Connectors

CCM (including Lightweight CCM) does not currently support explicit connector objects. Therefore, connectors are implicitly defined by the connections between components.

6.4.2 SDO

The Introspection package (Section 5.4) of the PIM relies on the Super Distributed Objects specification [SDO]. Implementations that support that package shall use that specification's CORBA PSM.

6.4.3 Behavior

This specification's PIM defines behavioral constraints that conforming implementations must respect. (For example, callbacks of data flow participant components must be invoked in a well-defined order, as described in Section 5.3.1). A middleware that hosts RT components and is implemented using CCM may choose how to satisfy those constraints. However, some non-normative suggestions are provided here.

- The execution semantics in this specification can be expressed in part using the Execution Models defined by the Streams for CCM Specification ([CCM STREAM], section 10.2).
- The implementation of a distributed execution context (in which RTCs on different nodes participate in the same context) may benefit from the *distributable thread* feature of the Real-Time CORBA specification [RT CORBA].

6.5 CORBA PSM

In this PSM, RTCs are mapped to CORBA interfaces extending the relevant IDL interfaces described in Section 6.1 and Section 6.2.

6.5.1 Mapping for Components

The mapping for IDL component-related concepts relies on the "equivalent interfaces" outlined in [CCM]. These IDL-to-IDL transformations remain normative.

The class generated for an IDL component shall not extend the CCM-specific base class **`Components::CCMObject`**. Instead, the class shall extend **`CORBA::Object`** as is typical of CORBA interfaces.

6.5.1.1 Mapping for Receptacles

6.5.1.1.1 Simplex Receptacles

A uses declaration of the following form:

uses <*interface_type*> <*receptacle_name*>;

results in the following equivalent operations defined in the component interface.

ReturnCode_t connect_<*receptacle_name*>(
in <*interface_type*> conxn);
<*interface_type*> disconnect_<*receptacle_name*>();
<*interface_type*> get_connection_<*receptacle_name*>();

These definitions match those defined by the Lightweight CCM specification except with respect to error reporting. They report error conditions using the `ReturnCode_t` mechanism defined in this specification rather than with explicit exceptions. The disconnect operation shall indicate failure with a nil return result.

6.5.1.1.2 Multiplex Receptacles

A uses declaration of the following form:

uses multiple <*interface_type*> <*receptacle_name*>;

results in the following equivalent operations defined in the component interface.

struct <*receptacle_name*>Connection {
<*interface_type*> objref;
RTC::Cookie ck;
};

sequence<<*receptacle_name*>Connection> <*receptacle_name*>Connections;

RTC::Cookie connect_<*receptacle_name*>(
in <*interface_type*> connection);
<*interface_type*> disconnect_<*receptacle_name*>(
in RTC::Cookie ck);
<*receptacle_name*>Connections get_connections_<*receptacle_name*>();

The above declarations are similar to those defined by the Lightweight CCM specification with two exceptions: error handling and the replacement of `Components::Cookie` with `RTC::Cookie`. The latter shall be identical to the former apart from the change in namespace.

The disconnect operation shall report errors by means of a nil return result.

6.5.2 Mapping for Connectors

CORBA does not currently support explicit connector objects. Therefore, connectors are implicitly defined by the references between objects.

6.5.3 SDO

The Introspection package (Section 5.4) of the PIM relies on the Super Distributed Objects specification [SDO]. Implementations that support that package shall use that specification's CORBA PSM.

6.5.4 Behavior

This specification's PIM defines behavioral constraints that conforming implementations must respect. (For example, callbacks of data flow participant components must be invoked in a well-defined order, as described in Section 5.3.1). A middleware that hosts RT components and is implemented using CORBA may choose how to satisfy those constraints. However, some non-normative suggestions are provided here.

- The implementation of a distributed execution context (in which RTCs on different nodes participate in the same context) may benefit from the distributable thread feature of the Real-Time CORBA specification [RT CORBA].

Annex A RTC IDL

(non-normative)

The normative IDL used by the PSMs in this specification is contained in the file RTC.idl that accompanies this document. A non-normative copy of that file's contents is provided here for convenience.

```
// RTC.idl

#include "SDOPackage.idl"

#pragma prefix "omg.org"

#define EXECUTION_HANDLE_TYPE_NATIVE long
module RTC {
    typedef EXECUTION_HANDLE_TYPE_NATIVEExecutionContextHandle_t;
    typedef SDOPackage::UniqueIdentifier         UniqueIdentifier;
    typedef SDOPackage::NVList                   NVList;

    enum ReturnCode_t {
        RTC_OK,
        RTC_ERROR,
        BAD_PARAMETER,
        UNSUPPORTED,
        OUT_OF_RESOURCES,
        PRECONDITION_NOT_MET
    };

    enum LifeCycleState {
        CREATED_STATE,
        INACTIVE_STATE,
        ACTIVE_STATE,
        ERROR_STATE
    };

    interface ExecutionContext;
    typedef sequence<ExecutionContext> ExecutionContextList;

    interface ComponentAction {
        ReturnCode_t on_initialize();
        ReturnCode_t on_finalize();
        ReturnCode_t on_startup(
            in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_shutdown(
            in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_activated(
            in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_deactivated(
```

};

```
            in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_aborting(
            in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_error(in ExecutionContextHandle_t exec_handle);
        ReturnCode_t on_reset(in ExecutionContextHandle_t exec_handle);
    };

    interface LightweightRTObject : ComponentAction {
        ReturnCode_t initialize();
        ReturnCode_t finalize();
        boolean is_alive(in ExecutionContext exec_context);
        ReturnCode_t reset();
        ReturnCode_t exit();
        ExecutionContextHandle_t attach_context(
            in ExecutionContext exec_context);
        ReturnCode_t detach_context(
            in ExecutionContextHandle_t exec_handle);
        ExecutionContext get_context(
            in ExecutionContextHandle_t exec_handle);
        ExecutionContextList get_owned_contexts();
        ExecutionContextList get_participating_contexts();
    };

    enum ExecutionKind {
        PERIODIC,
        EVENT_DRIVEN,
        OTHER
    };

    interface ExecutionContext {
        boolean is_running();
        ReturnCode_t start();
        ReturnCode_t stop();
        double get_rate();
        ReturnCode_t set_rate(in double rate);
        ReturnCode_t add_component(
            in LightweightRTObject comp);
        ReturnCode_t remove_component(
            in LightweightRTObject comp);
        ReturnCode_t activate_component(
            in LightweightRTObject comp);
        ReturnCode_t deactivate_component(
            in LightweightRTObject comp);
        ReturnCode_t reset_component(
            in LightweightRTObject comp);
        LifeCycleState get_component_state(
            in LightweightRTObject comp);
        ExecutionKind get_kind();
    };

    interface DataFlowComponentAction {
```

```
    ReturnCode_t on_execute(
        in ExecutionContextHandle_t exec_handle);
    ReturnCode_t on_state_update(
        in ExecutionContextHandle_t exec_handle);
    ReturnCode_t on_rate_changed(
        in ExecutionContextHandle_t exec_handle);
};

interface DataFlowComponent : DataFlowComponentAction {
};

interface Fsm {
};

interface FsmParticipantAction {
    ReturnCode_t on_action(
        in ExecutionContextHandle_t exec_handle);
};

interface FsmParticipant : FsmParticipantAction {
};

interface Mode {
};

interface ModeCapable {
    Mode get_default_mode();
    Mode get_current_mode();
    Mode get_current_mode_in_context(
        in ExecutionContext exec_context);
    Mode get_pending_mode();
    Mode get_pending_mode_in_context(
        in ExecutionContext exec_context);
    ReturnCode_t set_mode(
        in Mode new_mode,
        in boolean immediate);
};

interface MultiModeComponentAction {
    ReturnCode_t on_mode_changed(
        in ExecutionContextHandle_t exec_handle);
};

interface MultiModeObject : ModeCapable,
                            MultiModeComponentAction {
};

interface RTObject;

enum PortInterfacePolarity {
    PROVIDED,
```

```
    REQUIRED
};

struct PortInterfaceProfile {
    string instance_name;
    string type_name;
    PortInterfacePolarity polarity;
};

typedef sequence<PortInterfaceProfile> PortInterfaceProfileList;

interface PortService;
typedef sequence<PortService> PortServiceList;
typedef sequence<RTObject> RTCList;

struct ConnectorProfile {
    string name;
    UniqueIdentifier connector_id;
    PortServiceList ports;
    NVList properties;
};

typedef sequence<ConnectorProfile> ConnectorProfileList;

struct PortProfile {
    string name;
    PortInterfaceProfileList interfaces;
    PortService port_ref;
    ConnectorProfileList connector_profiles;
    RTObject owner;
    NVList properties;
};

typedef sequence<PortProfile> PortProfileList;

struct ExecutionContextProfile {
    ExecutionKind kind;
    double rate;
    RTObject owner;
    RTCList participants;
    NVList properties;
};

typedef sequence<ExecutionContextProfile>
ExecutionContextProfileList;

interface FsmObject {
    ReturnCode_t send_stimulus(
        in string message,
        in ExecutionContextHandle_t exec_handle);
};
```

```
struct FsmBehaviorProfile {
    FsmParticipantAction action_component;
    UniqueIdentifier id;
};

typedef sequence<FsmBehaviorProfile> FsmBehaviorProfileList;
struct FsmProfile {
    FsmBehaviorProfileList behavior_profiles;
};

interface FsmService : SDOPackage::SDOService {
     FsmProfile get_fsm_profile();
     ReturnCode_t set_fsm_profile(in FsmProfile fsm_profile);
};

struct ComponentProfile {
    string instance_name;
    string type_name;
    string description;
    string version;
    string vendor;
    string category;
    PortProfileList port_profiles;
    RTObject parent;
    NVList properties;
};

typedef sequence<ComponentProfile> ComponentProfileList;

interface PortService : SDOPackage::SDOService {
    PortProfile get_port_profile();
    ConnectorProfileList get_connector_profiles();
    ConnectorProfile get_connector_profile(
        in UniqueIdentifier connector_id);
    ReturnCode_t connect(
        inout ConnectorProfile connector_profile);
    ReturnCode_t disconnect(in UniqueIdentifier connector_id);
    ReturnCode_t disconnect_all();
    ReturnCode_t notify_connect(
        inout ConnectorProfile connector_profile);
    ReturnCode_t notify_disconnect(
        in UniqueIdentifier connector_id);
};

interface ExecutionContextService :   ExecutionContext,
                                      SDOPackage::SDOService {
    ExecutionContextProfile get_profile();
};

typedef sequence<ExecutionContextService>
```

```
    ExecutionContextServiceList;

    interface RTObject : LightweightRTObject, SDOPackage::SDO {
        ComponentProfile get_component_profile();
        PortServiceList get_ports();
    };
};
```

INDEX