Date: April 2013

Asynchronous Method Invocation for CORBA Component Model

Version 1.0

OMG Document Number: formal/2013-04-01
Standard document URL: http://www.omg.org/spec/AMI4CCM/
Associated File(s): http://www.omg.org/spec/AMI4CCM/20120707
http://www.omg.org/spec/AMI4CCM/20120707/ami4ccm.idl

Copyright © 2013, Object Management Group, Inc.
Copyright © 2011-2012, Remedy IT

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

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

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

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™, MOF™, OMG Interface Definition Language (IDL)™, and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

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

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

OMG's Issue Reporting Procedure

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

Table of Contents

Preface iii

1 Scope 1

2 Conformance 1

3 Normative References 1

4 Terms and Definitions 1

5 Symbols 2

6 Additional Information 2

7 AMI4CCM 5

- 7.1 Introduction 5
- 7.2 Running Example 5
- 7.3 Async Operation Mapping 6
  - 7.3.1 Callback Model Signatures (sendc) 7
- 7.4 Exception Delivery 8
  - 7.4.1 CCM_AMI::ExceptionHolder interface 9
- 7.5 Type-Specific ReplyHandler Mapping 9
  - 7.5.1 ReplyHandler Operations for normal replies 9
  - 7.5.2 ReplyHandler operations for handling exceptions 10
  - 7.5.3 Example 11
- 7.6 AMI4CCM Connector 11
- 7.7 Enabling AMI4CCM 13
- 7.8 Deployment Support 14

Annex A - IDL 17

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. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

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

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

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

Modernization Specifications

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

- **CORBAServices**

- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

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

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

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

About this Specification

Overview of this Specification

This specification defines a standardized way to perform asynchronous method invocations for CORBA Component Model (AMI4CCM). We start with explaining the concept and give an example of the implied IDL. After that a detailed overview of all implied IDL rules and the AMI4CCM connector will be listed. We will finish with how AMI4CCM impacts the deployment of a CCM application.

Intended Audience

This specification is intended for component implementors that want to use a standardized mechanism to perform asynchronous operations with CCM.

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/report_issue.htm*.

1 Scope

This specification defines a mechanism to perform asynchronous method invocation for CCM (AMI4CCM).

2 Conformance

This specification defines one optional CCM conformance point. In order to claim AMI4CCM compliance a CCM implementation must support the following conformance points:

- Implement all implied IDL generation.
- Generate the AMI4CCM interface-specific connector fragment.

3 Normative References

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

- OMG CORBA v3.2 Part 1 Interfaces specification: formal/2011-11-01
- OMG CORBA v3.2 Part 3 Components specification: formal/2011-11-03
- OMG Deployment and Configuration of Component-based Distributed Applications specification: formal/2006-04-02

4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative reference and the following apply.

Component
A specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository.

Connector
Interaction entity between components. A connector is seen at design level as a connection between components and is composed of several fragments (artifacts) at execution level, to realize the interaction.

Container
Containers provide the run-time execution environment for CORBA component implementations A container is a framework for integrating transactions, security, events and persistence into a component's behavior at runtime.

Executor
The programming artifact(s) that supply the behavior of a component, connector, or a component home.

Extended Port
Consists of zero or more provided as well as zero or more required interfaces, i.e., closely resembling the UML2 specification of a port.

Facet
A distinct named interface provided by the component for client interaction. The primary vehicle through which a component exposes its functional application behavior to clients during normal execution.

Fragment
Artifact, part of the connector implementation. A fragment corresponds to one executor that can be deployed onto an execution node, co-localized with one component for which it supports the interaction provided by the connector.

Implied-IDL
IDL that gets implicitly defined based on the user IDL. The implied-IDL is used to avoid an explicit mapping of the API to any programming language.

Port
A surface feature through which clients and other elements of an application environment may interact with a component. The component model supports four basic kinds of ports: facets, receptacles, event sources, event sinks, and attributes.

Receptacle
A named connection point that describes the component's ability to use a reference supplied by some external agent.

Simplex Receptacle
A specialization of a receptacle that only allows a single connection at a given time.

5 Symbols

List of symbols/abbreviations.

CCM — CORBA Component Model

IDL — Interface Definition Language

OMG — Object Management Group

ORB — Object Request Broker

UML — Unified Modeling Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

None

6.2 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first read the *CORBA Component Model* specification that complements this.

Although the clauses are organized in a logical manner and can be read sequentially, this reference specification is intended to be read in a non-sequential manner. Consequently, extensive cross-references are provided to facilitate browsing and search.

6.3 Acknowledgments

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

- Remedy IT
- Northrop Grumman

7 AMI4CCM

7.1 Introduction

Asynchronous Method Invocations for CORBA Component Model (AMI4CCM) allow client components to make non-blocking requests onto a target component. AMI4CCM is treated as a client-side language mapping issue only. Server-side implementations are not required to change as from the server-side programmer's point of view all invocations can be treated identically regardless of their synchronicity characteristics.

Clients may, at any time, make either asynchronous or synchronous requests on the target. One model of asynchronous requests is supported: callback. In the *callback* model, the client passes a reference to a reply handler (a client-side local object implementation that handles the reply for a client request), in addition to the normal parameters needed by the request. The reply handler interface defines operations to receive the results of that request (including **inout** and **out** values, return values, and possible exceptions). The **ReplyHandler** is a local object that is implemented by the programmer as with any local object implementation.

AMI4CCM may be used in single- and multi-threaded applications. AMI4CCM calls may have any legal return type, parameters, and contexts. AMI4CCM operations do not raise exceptions. Rather, user and system exceptions are passed to the implemented type-specific **ReplyHandler**. The only system exception that can be thrown when calling the AMI4CCM operation is the **INV_OBJREF** system exception. When this exception is thrown with a completion status of **COMPLETED_NO**, the request has not been made. This clearly distinguishes exceptions raised by the server (which are returned via the **ReplyHandler**) from AMI4CCM internal exceptions that caused AMI4CCM to fail.

This specification focuses entirely on the static (typed) asynchronous invocations that are based on the interface that is the target of the operation. This specification describes the mapping for the generated asynchronous method signatures. It also describes the generated reply handlers that are passed to those async methods when the callback model is used. The AMI4CCM mapping contains an IDL to "implied-IDL" mapping, which defines the new interface, operations, and connector required to perform asynchronous invocations and receive the replies to these requests. In general, the implied-IDL is used to avoid explicitly mapping the AMI4CCM API to each of the currently supported languages.

When an AMI4CCM-enabled IDL code generator is run on an interface, the following is performed in addition to the processing specified in *CCM*:

- A local interface mapping is generated for a type-specific **ReplyHandler** from which the client application derives its **ReplyHandler** implementation.
- A new local interface is generated which contains the asynchronous request operations with signatures as specified in "Async Operation Mapping" on page 6. The implied-IDL is used entirely so that each individual supported language mapping need not be given for the asynchronous request operations.

In order to annotate that the interface has to be AMI4CCM enabled an **ami4ccm interface** pragma has to added to the interface in IDL (see Section 7.7).

7.2 Running Example

A running example is used throughout this specification to clarify the generation of the new typed asynchronous invocation stubs, and the new reply handling interfaces for receiving callback responses. The example features a simple stock portfolio manager interface. Most importantly, the interface includes operations that cover all cases of operation signature:

- attributes
- in arguments
- inout arguments
- out arguments
- return values
- user exceptions

All occurrences of this example throughout this specification are not normative and only for explanation purposes.

```
// Original IDL
#pragma ami4ccm interface "StockManager"
#pragma ami4ccm receptacle "Client::manager"

exception InvalidStock { string sym; };

interface StockManager {
    attribute string stock_exchange_name;

    void        set_stock(in string symbol, in double new_quote)
        raises(InvalidStock);
    void        remove_stock(in string symbol, out double quote)
        raises(InvalidStock);

    boolean     find_closest_symbol(inout string symbol);
    double      get_quote(in string symbol) raises(InvalidStock);
};

component Client {
    uses StockManager manager;
};
```

7.3 Async Operation Mapping

For each operation in an interface, corresponding callback method signatures are generated. Note that no callback asynchronous method signatures are generated for any operations or attributes of abstract interfaces. Even though oneway operations have no associated reply, under certain circumstances it may be useful and necessary to receive a reply (either normal or exceptional). For oneway operations an invocation should result in an immediate callback on its reply handler.

These signatures are described in implied-IDL, which is used to generate language-specific operation signatures. These operations do not raise user exceptions but they can (but are not required to) raise system exceptions. For explanatory purposes, the sub clauses below show the callback implied-IDL. Logically, the IDL compiler deals with async as if the IDL included both the original IDL and the implied IDL.

The normative file **ami4ccm.idl** as listed in Annex A - IDL is intended to be used by the implied IDL. The file shouldn't be explicitly included by the user.

7.3.1 Callback Model Signatures (sendc)

When the callback model is used, the client supplies a reply handler when making the asynchronous invocation. The interface's operations and attributes are mapped to implied-IDL operations with names prefixed by "**sendc_**." If this implied-IDL operation name conflicts with existing operations on the interface or any of the interface's base interfaces, "**ami_**" strings are inserted between "**sendc_**" and the original operation name until the implied-IDL operation name is unique.

7.3.1.1 Implied-IDL for Operations

The signature of the implied-IDL for a given IDL operation is:

- void return type, followed by;
- **sendc_<opName>** where **opName** is the name of the operation.

The async callback version takes the following arguments in order:

- An object reference to a type-specific **ReplyHandler** as described in "Type-Specific ReplyHandler Mapping" on page 9, with the parameter name **ami_handler**. If a nil **ReplyHandler** reference is specified when this operation is invoked, no response will be returned for this invocation. A system exception may be raised by CCM during evaluation of the request, but once **sendc** returns, no further results of the operation will be made available.
- Each of the **in** and **inout** arguments in the order that they appeared in the operation's declaration in IDL, all with a parameter attribute of **in** and with the type specifier and parameter name of the original argument.
- **out** arguments are ignored (i.e., are not part of the async signature).

The implied-IDL operation signature has a context expression identical to the one from the original operation (if any is present).

7.3.1.2 Implied-IDL for Attributes

The signature of the implied-IDL for the callback model getter and setter operations corresponding to an interface's attribute is as follows.

- Setter operations are only generated for attributes that are not defined readonly
- void return type, followed by the operation name, which to distinguish between the getter and setter operations for the attribute is given by either:
  - **sendc_get_<attributeName>** for reading the attribute value, where **attributeName** is the name of the attribute, or
  - **sendc_set_<attributeName>** for setting the attribute value, where **attributeName** is the name of the attribute that is not defined **readonly**.

The callback implied-IDL operations take the following arguments in order:

- An object reference of a type-specific **ReplyHandler** as described in "Type-Specific ReplyHandler Mapping" on page 9, with the parameter name **ami_handler**.
- The additional arguments for asynchronous implied-IDL operations for **attributes** are as follows:
  - For the attribute's generated **get** operation, there are no additional arguments.

- For the attribute's generated **set** operation, there is one additional argument, in **<attrType> attr_<attributeName>**, where **attrType** is the type of the attribute, and **attributeName** is the name of that attribute. The **set** operation is only generated for attributes that are not defined **readonly**.

7.3.1.3 Example

The following implied-IDL is generated from the interface definitions used in the running example:

```
// AMI implied-IDL including callback operations
// for original example IDL defined in Section 7.2
// Pragmas that have to be added to the original IDL defined in Section 7.2, for more information
// regarding the pragmas see Section 7.7
#pragma ami4ccm interface "StockManager"
#pragma ami4ccm receptacle "Client::manager"

local interface AMI4CCM_StockManagerReplyHandler;

local interface AMI4CCM_StockManager {

    // Async Callback operation Declarations
    void sendc_get_stock_exchange_name(
            in AMI4CCM_StockManagerReplyHandler ami_handler);
    void sendc_set_stock_exchange_name(
            in AMI4CCM_StockManagerReplyHandler ami_handler,
            in string attr_stock_exchange_name);

    void sendc_set_stock(
            in AMI4CCM_StockManagerReplyHandler ami_handler,
            in string symbol, in double new_quote);
    void sendc_remove_stock(
            in AMI4CCM_StockManagerReplyHandler ami_handler,
            in string symbol);
    void sendc_find_closest_symbol(
            in AMI4CCM_StockManagerReplyHandler ami_handler,
            in string symbol);
    void sendc_get_quote(
            in AMI4CCM_StockManagerReplyHandler ami_handler,
            in string symbol);
};
```

7.4 Exception Delivery

The **ReplyHandler** interface is expressed in IDL and cannot have operations that take exceptions as arguments. Furthermore, the most natural way for a **ReplyHandler** to deal with exceptions is by invoking an operation that raises exceptions, not through inspecting operation parameters. Therefore, exception replies are propagated to the **CCM ReplyHandler** in the form of a type-specific **CCM_AMI::ExceptionHolder** interface that contains the marshaled exception as its state and has **raise_exception** operation for raising the encapsulated exception.

7.4.1 CCM_AMI::ExceptionHolder interface

The **CCM_AMI::ExceptionHolder** interface encapsulates the exception data and enough information to turn that data back into a raised exception. This interface must be defined in the file **ami4ccm.idl** that is listed in Annex A - IDL.

```
// IDL
module CCM_AMI {
    native UserExceptionBase;
    local interface ExceptionHolder {
        void raise_exception() raises (UserExceptionBase);
    };
};
```

- **raise_exception()** - This method is used by applications to raise the exception from the encapsulated **marshaled_exception** member.
- **UserExceptionBase** - Language mapping of this native type should allow any user exception to be raised from this method. For instance, it is mapped to CORBA::UserException in C++ and to org.omg.CORBA.UserException in java. As usual, system exceptions do not need to be in the raises clause for raising them from this method.

7.5 Type-Specific ReplyHandler Mapping

For each interface, a type-specific reply handler is generated by the IDL compiler. The client application implements and registers a reply handler with each asynchronous request and receives a callback when the reply is returned for that request. The interface name of the type-specific handler is **AMI4CCM_<ifaceName>ReplyHandler**, where **ifaceName** is the original unqualified interface name. If the interface **ifaceName** derives from some other IDL interface **baseName**, then the handler for **ifaceName** is derived from **AMI4CCM_<baseName>ReplyHandler**; otherwise it is derived from the generic **CCM_AMI::ReplyHandler**. The interface **CCM_AMI::ReplyHandler** has to be defined in the file **ami4ccm.idl** as shown in Annex A - IDL. If the interface name **AMI4CCM_<ifaceName>ReplyHandler** conflicts with an existing identifier, uniqueness is obtained by inserting additional "**AMI_**" prefixes before the **ifaceName** until the generated identifier is unique.

When invoking an async operation, the client first constructs a local object for its **ReplyHandler** and then associates it with the request by passing the local object as an argument to the operation. The reply will be targeted to that **ReplyHandler** so that a single **ReplyHandler** instance can be supplied to multiple requests, the client can assign unique ids for each request if the application code needs to distinguish between each of these requests at a later time. Most commonly, the application needs to access information from the calling scope while in the scope of the callback. That information can be associated with the **ReplyHandler**'s id by the client application at the time of invocation. Obtaining the **ReplyHandler**'s id within the callback implementation allows that implementation to obtain any information previously associated with the original request.

7.5.1 ReplyHandler Operations for normal replies

For each operation declared in the interface, an operation with the following signature is defined on the generated reply handler:

- return type void, followed by
    - the name of the operation, followed by
        - arguments in the following order (all "in" parameters).

- If the original operation has a return value, the type returned by the operation declared in IDL with parameter named **ami_return_val**.
- Each inout/out **type** name and **argument** name as they were declared in IDL.

These operations do not raise any exceptions because they are never invoked by a client and have no client to respond to such an exception.

For an attribute with the name "attributeName," the following operations are generated on the reply handler: return type void**,** followed by **get_<attributeName>** for the getter (or **set_<attributeName>** for the setter operation if the attribute is not defined to be readonly). For the "get" operation, there is one argument (the setter callback operation takes no arguments): **in <attrType> ami_return_val** where the attribute of name **ami_return_val** is of type **attrType**.

There are two cases where the above mapping results in an operation with no parameters. The first is for an operation with no return value and either with no parameters or with only **in** parameters. The second is the mapping of a setter on an attribute. In these cases, it is worth noting that the only meaning that can be associated with the operation is that the AMI4CCM operation completed successfully. This is significant information, essentially an acknowledgment of completion.

7.5.2 ReplyHandler operations for handling exceptions

If the AMI4CCM invocation didn't succeed at the target, the exception is delivered via the generated **_excep ReplyHandler** operation corresponding to the operation originally invoked. This sub clause describes the implied-IDL rules for generating these operations on the **ReplyHandler**.

For each operation, **operName**, on the original interface named **ifaceName**, an operation with the following signature is generated on the type-specific **ReplyHandler**:

```
void <operName>_excep(
    in CCM_AMI::ExceptionHolder excep_holder);
```

For each attribute, **attrName**, on the original interface named **ifaceName**, an operation with the following signature is generated on the type-specific **ReplyHandler**:

```
void get_<attrName>_excep(
    in CCM_AMI::ExceptionHolder excep_holder);
```

For each non-**readonly** attribute, **attrName**, on the original interface named **ifaceName**, an operation with the following signature is generated on the type-specific **ReplyHandler**:

```
void set_<attrName>_excep(
    in CCM_AMI::ExceptionHolder excep_holder);
```

If the name generated by the method described above clashes with a name that already exists in the interface, **"_ami"** strings are inserted immediately preceding the **"_excep"** repeatedly, until generated IDL operation name is unique in the interface.

7.5.3 Example

The example IDL causes the generation of the following additional IDL when asynchronous operations are to be used. This IDL is "real" in that the interfaces described here are local objects. The operations are invoked directly by the CCM infrastructure when a reply becomes available.

```
// AMI4CCM implied-IDL of ReplyHandler
// for original example IDL defined in  Section 7.2
local interface AMI4CCM_StockManagerReplyHandler : CCM_AMI::ReplyHandler {

    void get_stock_exchange_name(
        in string ami_return_val);
    void get_stock_exchange_name_excep(
        in CCM_AMI::ExceptionHolder excep_holder);

    void set_stock_exchange_name();
    void set_stock_exchange_name_excep(
        in CCM_AMI::ExceptionHolder excep_holder);

    void set_stock();
    void set_stock_excep(
        in CCM_AMI::ExceptionHolder excep_holder);

    void remove_stock(
        in double quote);
    void remove_stock_excep(
        in CCM_AMI::ExceptionHolder excep_holder);

    void find_closest_symbol(
        in boolean ami_return_val,
        in string symbol);
    void find_closest_symbol_excep(
        in CCM_AMI::ExceptionHolder excep_holder);

    void get_quote(
        in double ami_return_val);
    void get_quote_excep(
        in CCM_AMI::ExceptionHolder excep_holder);
};
```

7.6 AMI4CCM Connector

For each interface that is enabled for AMI4CCM using the **ami4ccm interface** pragma (see 7.7) an implied AMI4CCM connector has to be generated. The client will invoke at runtime the synchronous and asynchronous operations on this connector instead of to the real target component. In the file **ami4ccm.idl** the following templated connector must be defined (see Annex A - IDL). It is the responsibility of the AMI4CCM-aware code generation to generate a concrete AMI4CCM connector implementation for the AMI4CCM enabled interface.

```
module CCM_AMI
{
    // Base class for all AMI4CCM connectors
    connector AMI4CCM_Base
    {
    };

    // Templated Connector module for AMI4CCM. Expects
```

```
    // two template arguments, the original interface and
    // its AMI4CCM counterpart
    module Connector_T<interface T, interface AMI4CCM_T>
    {
        porttype AMI4CCM_Port_Type
        {
            // Deliver the asynchronous interface with its sendc_ operations
            provides AMI4CCM_T ami4ccm_provides;
            // Delivers the original interface for synchronous invocations through the connector
            provides T ami4ccm_sync_provides;
            // Use the port of the target component
            uses T ami4ccm_uses;
        };
        connector AMI4CCM_Connector : AMI4CCM_Base
        {
            port AMI4CCM_Port_Type ami4ccm_port;
        };
    };
};
```

For the given StockManager example the creation of a StockManager connector will be done through instantiating this IDL3+ templated module.

```
module CCM_AMI::Connector_T<StockManager, AMI4CCM_StockManager> AMI4CCM_StockManager_-
Connector;
```

Transforming the IDL3+ syntax to IDL2 will result in the following concrete local interface.

```
module AMI4CCM_StockManager_Connector
{
    local interface CCM_AMI4CCM_Connector
        : CCM_AMI::CCM_AMI4CCM_Base
    {
        ::CCM_AMI4CCM_StockManager get_ami4ccm_port_ami4ccm_provides ();
        ::CCM_StockManager get_ami4ccm_port_ami4ccm_sync_provides ();
    };
};
```

For the connector the following context will be generated.

```
module AMI4CCM_StockManager_Connector
{
    local interface CCM_AMI4CCM_Connector_Context
        : CCM_AMI::CCM_AMI4CCM_Base_Context
    {
        ::StockManager get_connection_ami4ccm_port_ami4ccm_uses ();
    };
};
```

7.7 Enabling AMI4CCM

The first step in enabling AMI4CCM is by specifying the **ami4ccm interface** pragma for the interface that needs to be enabled with AMI4CCM.

```
#pragma ami4ccm interface "<fully qualified interface name>"
```

The second step identifies the AMI4CCM enabled receptacles; as a component can have a huge number of receptacles and in most cases only a few of these receptacles are intended to be used with AMI4CCM. To enable AMI4CCM for a given receptacle, the component developer has to annotate the IDL to enable AMI4CCM for a specific receptacle. This has to be done using a pragma that needs to be specified for each AMI4CCM enabled receptacle.

```
#pragma ami4ccm receptacle "<fully qualified receptacle name>"
```

This is used by the AMI4CCM aware IDL compiler to generate an additional method in the context. The implied AMI4CCM receptacle will be named **sendc_<receptacle name>**. Enabling AMI4CCM for a given receptacle has impact on the generated context of the client component.

```
#pragma ami4ccm receptacle “Client::manager”
component Client
{
    uses StockManager manager;
};
```

Will result in the following context generated

```
local interface CCM_Client_Context
    : ::Components::SessionContext
{
    StockManager get_connection_manager ();
    // Implied method for AMI4CCM triggered by pragma
    AMI4CCM_StockManager get_connection_sendc_manager ();
};
```

In case of a multiplex receptacle the context will return a sequence of object references for AMI4CCM.

```
#pragma ami4ccm receptacle “StockManagerManager::managers”
component StockManagerManager
{
    uses multiple StockManager managers;
};
```

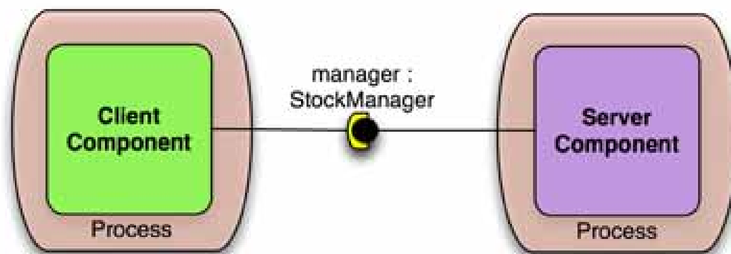
Will result in the following context, where sendc_StockManagers is an implied sequence.

```
local interface CCM_StockManagerManager_Context
    : ::Components::SessionContext
{
    StockManagers get_connections_managers;
    // Implied method for AMI4CCM triggered by pragma
    sendc_StockManagers get_connections_sendc_managers;
};
```

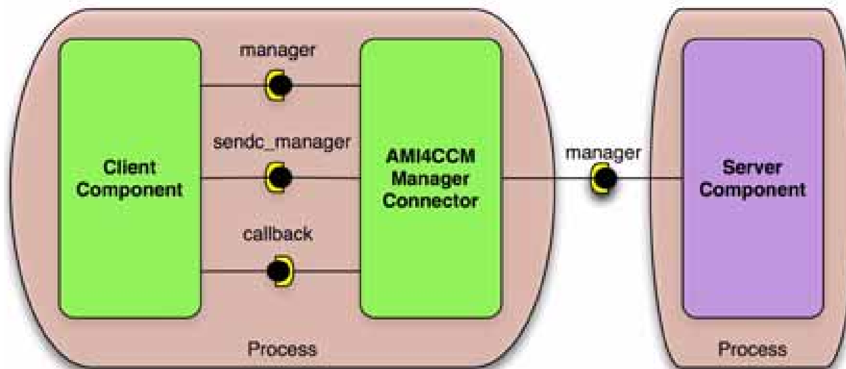
7.8 Deployment Support

At runtime for the AMI4CCM connector an AMI4CCM Connector fragment has to be deployed by the D&C infrastructure. This deployment will be done in compliance with the CCM (see 7.5 - Connector Deployment and Configuration). The AMI4CCM Connector fragment will be deployed like a normal component. Between the client component and the fragment the connection(s) will be created by the D&C, just as between the fragment and the receiver component.

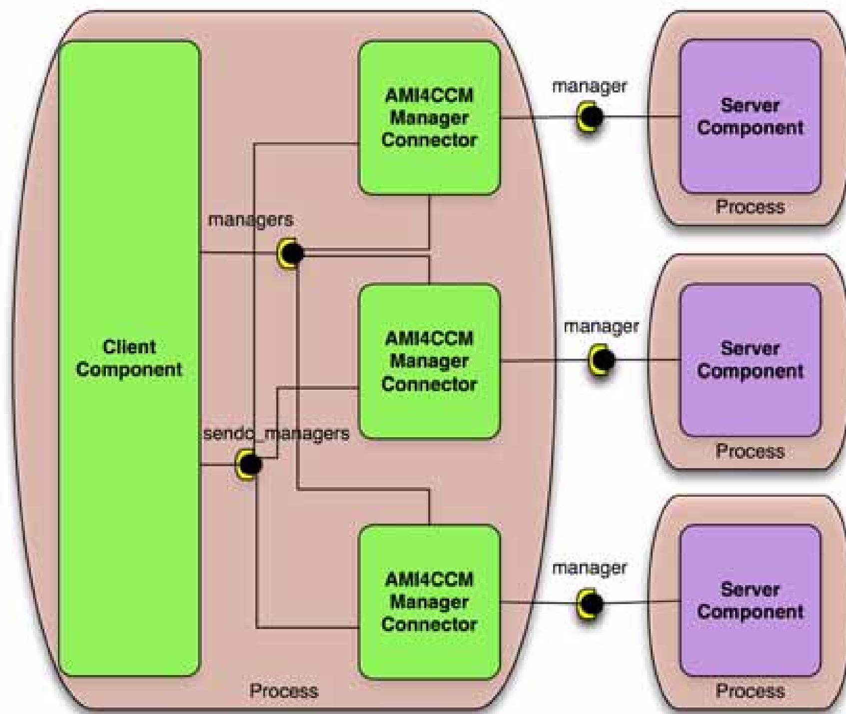
At a logical level the component developer has the following components with AMI4CCM enabled on the manager receptacle of the client component.



At deployment time this will lead to the following deployed components, connector fragment, and connections. The client component and fragment are required to be deployed within the same process.



In case a multiplex receptacle has been enabled for AMI4CCM, for each server component a connector fragment will be deployed collocated to the client. Between the connector fragment and the server component only a simplex connection can be defined.



Annex A - IDL

(normative)

ami4ccm.idl

```
#include <Components.idl>

module CCM_AMI {

    native UserExceptionBase;

    /// Exception holder to rethrow the original exception
    local interface ExceptionHolder {
        void raise_exception() raises (UserExceptionBase);
    };

    /// Base interface for the Callback model
    local interface ReplyHandler {
    };

    /// Base class for all AMI4CCM connectors
    connector AMI4CCM_Base {
    };

    /**
    * Templated Connector module for AMI4CC. Expects
    * two template arguments, the original interface and
    * its AMI4CCM counterpart
    */
    module Connector_T<interface T, interface AMI4CCM_T> {

        porttype AMI4CCM_Port_Type {
            provides AMI4CCM_T ami4ccm_provides;
            provides T ami4ccm_sync_provides;
            uses T ami4ccm_uses;
        };

        connector AMI4CCM_Connector : AMI4CCM_Base {
            port AMI4CCM_Port_Type ami4ccm_port;
        };
    };
};
```