Light Weight CORBA Component Model

Revised Submission

Mercury

THALES

With support and collaboration from:
Raytheon

Table Of Contents

1 Introduction

1.1 Submitting Companies

The following companies are pleased to submit this specification in response to the Lightweight CCM Request for Proposal (realtime/2002-08-01):

Mercury Computer Systems, Inc.

THALES

1.2 Status of the Document

This document is the revised submission produced for the OMG Technical Committee meeting to be held in Paris in June 2003.

1.3 Guide to the Submission

This submission presents what functionality must be supported by a lightweight CCM, which represents a subset of the functionality of the full CORBA CCM specification. Lightweight CCM components must be as much as possible interoperable with “full” CCM components.

Chapter 1 contains the introduction to the proposal. Chapter 2 describes the requirements of a lightweight component model. Chapter 3 provides the rules, principles and rationale. Chapter 4 follows the “full” CCM specification chapter by chapter and identifies which parts are included in the Lightweight CCM subset.

1.4 Submission Contact Points

Olivier Hachet
THALES Communications
1-5, avenue Carnot – 91883 Massy Cedex – France
Tel: + 33 (0) 1 69 75 32 39 Fax: + 33 (0) 1 69 75 31 79
mailto: olivier.hachet2@fr.thalesgroup.com

James Kulp
Mercury Computer Systems, Inc.
199 Riverneck Road, Chelmsford, MA, 01824-2820, U.S.A.
Tel: + 1 (978) 256-0052 x1251
mailto: jek@mc.com

Frank Pilhofer
Mercury Computer Systems, Inc.
199 Riverneck Road, Chelmsford, MA, 01824-2820, U.S.A.
Tel: + 1 (978) 256-0052 x3129
mailto: fp@mc.com

2 Problem to be Solved

Many of today's embedded CORBA applications are unable to use the available enterprise CCM due to design constraints imposed on them by their operational environment. These constraints include small code size in embedded environments and limited processing overhead for performance conservative applications. These constrained environments need CCM functionality tailored to create a reduced functionality or "lightweight" version. ORB vendors, or other third-party vendors, could then support this lightweight version.

In some cases, vendors are currently supplying special lightweight, but non-standard CCM versions. An example of this would be the XML profiles used for deploying waveform applications in a software radio such as the Joint Tactical Radio System. However, application developers that use those special lightweight component deployment services cannot rely upon a common subset of features for their applications. This profile provides an opportunity to create a standard specification where one is certainly needed.

There are three areas where the CCM can be profiled for use with these constrained systems: capabilities, interfaces, and XML. Reducing the scope in these areas will result in the creation of a lightweight version of the CCM. The purpose of this profile is to specify a lightweight version of the CCM.

This profile tries to be as compliant as possible to the OMG "*Minimum CORBA*" specification (*Chapter 23, CORBA 2.6, 01-12-61*) as indicated in the "*Lightweight CCM Request for Proposal*".

3 Methodology

This document follows each chapter of the CCM specification. Each of them are treated in Chapter 4 of this specification except Chapter 8 (*Interface Repository MetaModel*) and chapter 9 (*CIF Metamodel*) of the "full" CCM. This profile is based on the CORBA Component specification (*formal/02-06-65*).

Each chapter of the CCM specification is analyzed to extract what is relevant to be included in the Lightweight CCM profile.

The goal is to simplify as much as possible the use of CCM for embedded applications and to avoid presenting the user with multiple ways to do the same thing.

The principal aim is to have a component model sufficient to compose applications with CORBA components without all optional features that are not part of the "*core*" capabilities CCM, which are defined as:

- A component model allowing definition of components with provided and needed services (interfaces and events).
- The possibility to interconnect components via these ports.
- A container programming model
  - to specify how components interact with their environment (other than the defined interfaces)
  - to clearly separate the functional properties (provided by the component itself) from the non-functional properties (that are, in return, fulfilled by its hosting ***container*** with the help of the underlying execution platform).
- The possibility to reuse components in many applications

This profile exposes what mandatory features should be contained in a minimum implementation of the CCM. It doesn't preclude vendors to implement more than these features.

The choices made in the profile follow some rules established to suit embedded environments. These rules are the following:

1. Redundancy: If several ways of requesting a service exist, only one is retained. A priority is given to minimize the size of generated code.

2. Interoperability and compatibility with "full" CCM: Interoperability between CCM components and lightweight components should follow these rules:
   - During deployment, a lightweight component should be deployable by a "full" CCM deployment application.
   - A lightweight component does not have to be supported in a "full" CCM container nor the opposite.
   - Connections between a lightweight component and a "full" CCM component must be possible.
   - Implementations of lightweight components should be source compatible with the "full" CCM.

3. Persistence: The lightweight CCM does not need to manage any kind of persistence as described in the CCM specification. This point is developed in section 4.2.

4. Transactions: Transactions are not a feature commonly used in embedded systems targeted by this profile and thus are not included.

5. Security: Security will not be treated in this profile.

6. Introspection: Not all introspection operations are retained in this profile because they are not essential to perform the deployment of components and this information is generally known at deployment time by the deployment system.

7. valuetypes: This profile is based on the Minimum CORBA specification where *valuetypes* are not excluded.

4 CORBA Component Model Review

4.1 Component Model - CCM Chapter 1

4.1.1 Component Model

In the section 1.1 of the CCM specification, the components types are specified in IDL and represented in the Interface Repository. Because most of the Interface Repository is omitted from Minimum CORBA, lightweight components types will not be represented in the Interface Repository. Therefore, no operations are planned on the interfaces of a lightweiht component In embedded applications, there is generally no need to have an interface data base.

4.1.1.1 Component Levels

This specification will support both the component levels: *basic* and *extended*, since basic components were defined to align with JavaBeans, and forego many fundamental utilities of components that are valuable to embedded systems. However, there is no separate compliance point for basic components in this profile.

4.1.1.2 Ports

All the kind of ports (Facets, Receptacles, Event Sources, Event Sinks and Attributes) are pertinent for a component model for embedded applications and are therefore retained.

4.1.1.3 Components and Facets

The whole section is retained in the profile.

4.1.1.4 Component Identity

A component instance is identified primarily by its component reference, and secondarily by its set of facet references (if any).

In this profile, components cannot be associated with *primary keys* as it is defined in the CCM specification, because homes managing primary keys are omitted from the lightweight CCM (see chapter 4.1.7). Thus primary keys are excluded from the profile.

4.1.1.5 Component Homes

Because the lightweight CCM approach is to simplify CCM and suppress redundant functionality, the use of keyless homes is sufficient for these applications. This avoids the management of home lists in the application.

In addition, keyless homes are simplified. Being a factory of components is the essential interest of homes. Therefore only one kind of homes is retained (see section 4.1.7).

4.1.2 Component Definition

The whole section is retained in the profile.

4.1.3 Component Declaration

The sections concerning "*Basic Components*", "*Equivalent IDL*" and "*Component Body*" are retained in the profile.

4.1.4 Facets and Navigation

The facet concept as it is described in section 1.4 of the CCM specification is retained in this profile, including the keywords **component**, **supports**, **provides**.

On the other hand, the navigation is simplified. In embedded systems, this feature is not essential and only used for the deployment of components (more precisely connection). Most of the introspection methods are excluded from this profile.

In general, the generic operations are preferred to the typed ones even if they do not offer type safety. They have two advantages:

- they can be implemented with a smaller footprint than the implied-IDL ones
- they are a convenient way to be compliant with a CCM deployment tool.

The specific operations to access provided interfaces (**provide_<name>**) are redundant with the generic one called **provide_facet**, so they are excluded.

In this profile, interface **Navigation** only contain the **provide_facet** interface which is needed for the connection phase. Therefore,

PortDescription, **FacetDescription**, **get_all_facets**, **get_named_facets** and **same_component** are excluded.

Here is the **Navigation** interface for lightweight CCM:

```
module Components {
        typedef string FeatureName;
        typedef sequence<FeatureName> NameList;
        valuetype PortDescription
        {
                public FeatureName name;
                public CORBA::RepositoryId type_id;
        };
        valuetype FacetDescription : PortDescription
        {
                public Object facet_ref;
        };
        typedef sequence<FacetDescription> FacetDescriptions;
        exception InvalidName { };
        interface Navigation {
                Object provide_facet (in FeatureName name)
                raises (InvalidName);
                FacetDescriptions get_all_facets();
                FacetDescriptions get_named_facets (in NameList names)
                raises (InvalidName);
                boolean same_component (in Object object_ref);
        };
};
```

4.1.5 Receptacles

Receptacle concept as described in section 1.5 of the CCM specification, is retained in this profile, including keywords **uses** and **multiple**.

On the other hand, the implied specific connection operations are excluded: only the generic operations are retained, for the same reasons as the section 4.1.4.

4.1.5.1 Equivalent IDL

Following the general rules, the operations **connect_<receptacle_name>**, **disconnect_<receptacle_name>** and **get_connections_<receptacle_name>** are excluded because only generic connection operations are retained.

4.1.5.2 Behavior

Port-specific connection generated methods are excluded since generic operations exist in the receptacle interface.

Simplex and *multiplex* receptacles are retained.

4.1.5.3 Receptacle Interface

For the same reasons explained in 4.1.4, **get_connections** will be excluded from the **Receptacles** interface which implies exclusion of **ConnectionDescription** *valuetype*, **get_all_receptacles** and

get_named_receptacles also which imply exclusion of **ReceptacleDescription** *valuetype*.

Receptacle interface is now as follow:

```
module Components {
        valuetype ConnectionDescription {
                public Cookie ck;
                public Object objref;
        };
        typedef sequence<ConnectionDescription> ConnectionDescriptions;
        valuetype ReceptacleDescription : PortDescription
        {
                public boolean is_multiple;
                public ConnectionDescriptions connections;
        };
        typedef sequence<ReceptacleDescription> ReceptacleDescriptions;

        exception ExceededConnectionLimit { };
        exception CookieRequired { };

        interface Receptacles {
                Cookie connect ( in FeatureName name, in Object connection )
                raises ( InvalidName,
                        InvalidConnection,
                        AlreadyConnected,
                        ExceededConnectionLimit);
                void disconnect (in FeatureName name, in Cookie ck)
                raises ( InvalidName,
                        InvalidConnection,
                        CookieRequired,
                        NoConnection);
                ConnectionDescriptions get_connections (in FeatureName
                name) raises (InvalidName);
                ReceptacleDescriptions get_all_receptacles ();
                ReceptacleDescriptions get_named_receptacles (in NameList
                names) raises(InvalidName);
        };
};
```

4.1.6 Events

The CCM supports a publish/subscribe event model, compatible with CORBA Notification Service. The aim of this section is to retain the sufficient features to be compliant with the "*Lightweight Event Service*" specified in the emerging OMG specification for "*Lightweight Services"*.

This specification is based on the CORBA Event Service and only the push model is retained and it is fully compatible with the "full" CORBA Event Service. Since the CCM events are also based on the push model, references to the Notification Service have to be replaced by references to the Event Service.

4.1.6.1 Event types

Since the underlying implementation of the component event mechanism provided by the container is based on "*Lightweight Event Service*", event values shall be inserted into instances of any type. The mapping between a component event and a lightweight event is implemented by the container.

The whole sub-chapters "*Equivalent IDL*" and "*EventBase*" are retained.

4.1.6.2 EventConsumer Interface

The whole chapter is retained.

4.1.6.3 Event Service Provided by Container

The whole chapter is retained.

4.1.6.4 Event Sources-Publishers and Emitters

The two categories of event sources, *emitters and publishers* are retained.

4.1.6.5 Publisher

The operations **subscribe_<source_name>** and **unsubscribe_<source_name>** to respectively connect / disconnect the consumer parameter to / from an event channel are excluded because equivalent generic operations **subscribe** and **unsubscribe** of the **Events** interface are retained.

4.1.6.6 Emitters

The operations **connect_<source_name>** and **disconnect_<source_name>** to respectively connect / disconnect the consumer parameter to / from an event emitter are excluded because equivalent generic operations **connect_consumer** and **disconnect_consumer** of the **Events** interface are retained.

4.1.6.7 Event Sinks

This chapter is retained except for the **get_consumer_<sink_name>** operation which is redundant with the generic operation **get_consumer** of the **Events** interface.

4.1.6.8 Events Interface

Introspection operations are excluded.

The **Events** interface is now described as follows:

```
module Components {
        exception InvalidName { };
        exception InvalidConnection { };
        exception AlreadyConnected { };
        exception NoConnection { };
        valuetype ConsumerDescription : PortDescription
        {
                public EventConsumerBase consumer;
        };
        typedef sequence<ConsumerDescription> ConsumerDescriptions;
        valuetype EmitterDescription : PortDescription
        {
```

```
            public EventConsumerBase consumer;
        };
        typedef sequence<EmitterDescription> EmitterDescriptions;
        valuetype SubscriberDescription
        {
            public Cookie ck;
            public EventConsumerBase consumer;
        };
        typedef sequence<SubscriberDescription> SubscriberDescriptions;
        valuetype PublisherDescription : PortDescription
        {
            public SubscriberDescriptions consumers;
        };
        typedef sequence<PublisherDescription> PublisherDescriptions;
        interface Events {
            EventConsumerBase get_consumer (in FeatureName sink_name)
            raises (InvalidName);

            Cookie subscribe (in FeatureName publisher_name,
            in EventConsumerBase subscriber) raises (InvalidName,
            InvalidConnection, ExceededConnectionLimit);

            void unsubscribe (in FeatureName publisher_name,
            in Cookie ck) raises (InvalidName, InvalidConnection);

            void connect_consumer (in FeatureName emitter_name, in
            EventConsumerBase consumer) raises (InvalidName,
            AlreadyConnected, InvalidConnection);
            EventConsumerBase disconnect_consumer (in FeatureName
            source_name) raises (InvalidName, NoConnection);
            ConsumerDescriptions get_all_consumers ();
            ConsumerDescriptions get_named_consumers ( in NameList
            names) raises (InvalidName);
            EmitterDescriptions get_all_emitters ();
            EmitterDescriptions get_named_emitters (in NameList names)
            raises (InvalidName);
            PublisherDescriptions get_all_publishers ();
            PublisherDescriptions get_named_publishers (in NameList
            names) raises (InvalidName);
        };
};
```

4.1.7 Homes

The essential function of homes is to be a component factory and this seems sufficient for embedded applications.

CCMHome interface is retained to insure compatibility with existing CCM, but only essential operations are retained.

As it is expressed in section 4.1.1.5, the use of keyless homes is sufficient for lightweight components. Keyless homes are retained, but simplified. Keyed homes are excluded.

Home definitions are described in IDL in section 4.1.7.1.

4.1.7.1 Equivalent Interfaces

To avoid redundancy concerning ways to create components, only home definition with no primary key is retained. A simple way to manage a home is to only use the home declaration in the IDL3 component definition.

```
home <home_name> manages <component_type> {
        <explicit_operations>
};
```

Even if **<explicit_operations>** are redundant with implicit operations, they are retained because there is no additional cost if the component developer uses it. And it can be useful to have user defined homes (for instance, to initialize read-only attributes of the components at creation time).

The resulting interfaces for the home definitions with no primary key have the following forms:

```
interface <home_name>Explicit : Components::CCMHome {
        <equivalent_explicit_operations>
};
interface <home_name>Implicit : Components::KeylessCCMHome {
        <component_type> create() raises(CreateFailure);
};
interface <home_name> : <home_name>Explicit, <home_name>Implicit { };
```

The **create** operation creates a new component instance of the type managed by the home.

The "*Home definitions with primary keys*" sub-chapter is not retained.

In embedded applications, homes do not need to have specific behavior, therefore, the "*Supported Interfaces*" section is excluded.

As homes with primary key and supported interfaces are not mandatory in the original CCM specification, these restrictions have no impact on the "full" CCM compliance.

4.1.7.2 Primary Key Declaration and Explicit Operations

Chapter 1.7.2 and 1.7.3 of the CCM specification are excluded from this profile.

4.1.7.3 Home Inheritance

Homes as defined in this profile are light and simple. Thus they do not need to inherit from one another since their use is limited to component creation and possibly configuration.

4.1.7.4 Semantics of Home Operations

Orthodox operations (without *inheritance* and *primary keys* problems) are retained in this profile. If necessary, the user can define factory operations on homes. These *heterodox* operations are restricted to factory operations without inheritance and primary keys.

4.1.7.5 CCMHome Interface

The **CCMHome** interface becomes as follow:

```
module Components {
        typedef unsigned long FailureReason;
        exception CreateFailure { FailureReason reason; };
        exception FinderFailure { FailureReason reason; };
        exception RemoveFailure { FailureReason reason; };
        exception DuplicateKeyValue { };
        exception InvalidKey { };
        exception UnknownKeyValue { };
        interface CCMHome {
                CORBA::IRObject get_component_def();
                CORBA::IRObject get_home_def ();
                void remove_component ( in CCMObject comp)
                raises (RemoveFailure);
        };
};
```

```
module Components {
   interface KeylessCCMHome {
      CCMObject create_component() raises (CreateFailure);
   };
};
```

get_component_def and **get_home_def** are excluded since they are introspection operations. Others are excluded because they are related to primary keys or home finders.

4.1.8 Home Finders

Embedded applications do not need home finders because in such environments, the number of homes will be limited and such a heavy and complex mechanism to find them is not justified. The use of the naming service will be sufficient for retrieving homes. Thus this section is excluded from this profile.

4.1.9 Component Configuration

This chapter is pertinent for embedded environments, and thus is retained.

It should be noted that the two following concerns deserve a peculiar attention in the case of embedded systems:

- the distinction to be made between interface features that are used primarily for configuration, and interface features that are used primarily by application clients during normal application operation.
- the exclusive configuration and operational life cycle phases which divide the component life cycle into two mutually exclusive phases:
  - configuration phase
  - operational phase.

4.1.10 Configuration with Attributes

The “full” CCM specification defines several mechanisms to configure components.

- The first one is the attribute **Configurator** which is an object that encapsulates an attribute. It provides a **configure** operation that can be applied to a given component. A **StandardConfigurator** interface

supports the ability to provide the configurator with a set of values. These values are passed via a parameter of the operation **set_configuration**.

- The second one is the *factory-based configuration*: it is based on an interface called **HomeConfiguration** which can be used with or without **Configurator**s**.** This one is simpler and sufficient to set configuration values and applied them to component instances.

Therefore, **Configurator** interface is excluded and **HomeConfiguration** is retained without the use of **Configurator**.

The **HomeConfiguration** interface is as follows:

```
module Components {
        interface HomeConfiguration : CCMHome {
                ~~void set_configurator (in Configurator cfg);~~
                void set_configuration_values (in ConfigValues config);
                void complete_component_configuration (in boolean b);
                void disable_home_configuration();
        };
};
```

The operation **set_configurator** is excluded because **Configurator**s are not retained.

4.1.11 Component Inheritance

Globally, the component inheritance principle is valid for environments targeted by lightweight CCM.

The figure 1-2 of the CCM specification is not modified.

4.1.11.1 CCMObject Interface

The **CCMObject** interface is now defined by the following IDL:

```
module Components {
    ~~valuetype ComponentPortDescription~~
    ~~{~~
        ~~public FacetDescriptions facets;~~
        ~~public ReceptacleDescriptions receptacles;~~
        ~~public ConsumerDescriptions consumers;~~
        ~~public EmitterDescriptions emitters;~~
        ~~public PublisherDescriptions publishers;~~
    ~~};~~
    ~~exception NoKeyAvailable { };~~
   interface CCMObject : Navigation, Receptacles, Events {
        ~~CORBA::IRObject get_component_def ( );~~
        CCMHome get_ccm_home( );
        ~~PrimaryKeyBase get_primary_key( ) raises (NoKeyAvailable);~~
         void configuration_complete( ) raises (InvalidConfiguration);
        void remove() raises (RemoveFailure);
        ~~ComponentPortDescription get_all_ports ();~~
    };
};
```

get_primary_key, **get_component_def, get_all_ports** (introspection operations) and **ComponentPortDescription** are excluded.

4.1.12 Conformance Requirements

This section identifies the conformance points required for compliant implementations of the lightweight CCM. The points expressed below resume the points of the "full" CORBA Component model. Modification on existing points and new points are in bold characters. New points are added behind the existing "full" CCM points.

1. A CORBA COS vendor shall provide the relevant changes to the Lifecycle, Transaction, and Security Services identified in the following Section 1.12.2, "Changes to Object Services," on page 1-55 (*CCM specification*).

2. A CORBA ORB vendor need not provide implementations of Components aside from the changes made to the Core to support components. Conversely a CORBA Component vendor need not be a CORBA ORB vendor.

3. A CORBA Component vendor shall provide a conforming implementation of the Basic Level of CORBA Components.

4. A CORBA Component vendor may provide a conforming implementation of the Extended Level of CORBA Components.

5. To be conformant at the Basic level a non-Java product shall implement (at a minimum) the following:

   - the IDL extensions and generation rules to support the client and server side component model for basic level components.
   - CIDL. The multiple segment feature of CIDL (Section 2.12, "Segment Definition," on page 2-10) need not be supported for basic components.
   - a container for hosting basic level CORBA components.
   - ~~the XML deployment descriptors and associated zip files for basic components in the format defined in Section 6.1, "Introduction," on page 6-1~~.
   - **the XML deployment descriptors and associated zip files format for basic components will be defined in the OMG "Deployment and Configuration " specification (mars/2003-03-04).**

   Such implementations shall work on a CORBA ORB as defined in #1 above.

6. To be conformant at the Basic level a Java product shall implement (at a minimum):

   - ~~EJB1.1, including support for the EJB 1.1 XML DTD~~.
   - the java to IDL mapping, also known as RMI/IIOP.
   - ~~EJB to IDL mapping as defined in Section 5.3.2, "Translation of CORBA Component requests into EJB requests," on page 5-9~~.

   Such implementations shall work in a CORBA interoperable environment, including interoperable support for IIOP, CORBA transactions and CORBA security.

7. To be conformant at the extended level, a product shall implement (at a minimum) the requirements needed to achieve Basic PLUS:

- IDL extensions to support the client and server side component model for extended level components.
- A container for hosting extended level CORBA components.
- ~~The XML deployment descriptors and associated zip files for basic and enhanced level components in the format defined in Section 6.1, "Introduction," on page 6-1~~.
- **The XML deployment descriptors and associated zip files format for basic components will be defined in the OMG " Deployment and Configuration " specification (mars/2003-03-04).**

Such implementations shall work on a CORBA ORB as defined in #1 above.

8. ~~A CORBA Component vendor may optionally support EJB clients interacting with CORBA Components, by implementing the IDL to EJB mapping as defined in Section 5.4.2, "Translation of EJB requests into CORBA Component Requests," on page 5-17.~~

9. This specification includes extensions to IDL, in the form of new keywords and grammar. Although a CORBA ORB vendor need not be a CORBA Component vendor, and vice-versa, it is important to maintain IDL as a single language. To this end, all compliant products of any conformance points above shall be able to parse any valid IDL definitions. However, it is permitted to raise errors, or to ignore, those parts of the grammar that relate to another conformance point.

10. **A CORBA Component vendor shall implement at minimum the whole features of the lightweight CCM.**

11. **Lightweight CCM implementation shall be compliant with a "full" CCM implementation on the following aspects:**

    - **A lightweight component should be deployable by a "full" CCM deployment application.**
    - **Connections between a lightweight component and a "full" CCM component must be possible.**
    - **Implementations of lightweight components should be source compatible with the "full" CCM.**
    - **A lightweight component does not have to be supported in a "full" CCM container nor the opposite.**

Conforming implementations as defined above may also implement any additional features of this specification not required by the above conformance points.

Concerning tools, the same requirements are valid for lightweight CCM.

Changes to object services section has no impact one the current proposal.

4.2 OMG CIDL Syntax and Semantics - CCM Chapter 2

In the CCM specification, the unit of implementation is the *composition*. It is defined by CIDL files. A composition definition is a named scope that

contains elements that constitute the composition. The elements of a composition definitions are as follows:

- The keyword **composition**.
- The specification of the life cycle category, one of the keywords **service**, **session**, **process**, or **entity**.
- An identifier that names the composition in the enclosing module scope.
- The composition body. The composition body consists of the following elements:
  - an optional catalog usage declaration,
  - a mandatory home executor definition, and
  - an optional proxy home definition.

All these definitions can be retrieved in IDL3 declarations or defined in XML descriptor files if necessary. The ways to do it are detailed in the next section.

Finally, the whole chapter concerning CIDL is excluded from this profile.

See the next section for rationale for these exclusions.

4.3 CCM Implementation Framework - CCM Chapter 3

4.3.1 Introduction

The CIF for the Lightweight CCM will use the IDL3 descriptions and possibly an XML description file of the component (like **.ccd** files specified in the "full" CCM specification) to generate programming skeletons.

4.3.2 CIF Architecture

The CIF is designed to be compatible with the existing POA framework by using its features but not exposing them.

4.3.2.1 Component Implementation Definition Language

CIDL is the focal point of the CIF in the "full" CCM specification, and it permits to describe components structure and to generate the associated programming skeletons.

In our case, the CIDL part of the CCM specification has been removed, so the code generation has to be based on other information. This information already exists or can be described in an other way.

The CIDL is redundant with IDL3 definitions because all functional descriptions of the component is done with the OMG IDL3 files. These descriptions are

- the provided interfaces (facets),
- the used interfaces (receptacles),
- the events sources,

- the event sinks,
- the attributes.

The CIDL provides other information not present in IDL3 definition. These information are:

- component persistence and behavior,
- component category (Service, Session, Process, Entity)
- segmentation of the component

Concerning persistence, it is not in the scope of this document (see Methodology chapter 3).

The way to assign a component category (**service** or **session**) to a component, can be done via an XML description file that will be used with the IDL3 files to generate container code and skeletons. There is no need of CIDL language for that purpose. An example could be the use of a descriptor file like the **.ccd** file which is in the "full" CCM, partially generated from CIDL declarations. This file can be for this profile, written by the user.

To simplify the existing CCM and to fit low footprint requirements, this profile is only interested in "*monolithic*" components; this means that the notion of segmentation is not taken into account. This concept has an interest only when component have to be segmented because their provided interfaces have different behavior or persistence state. It is therefore not in the scope of this profile.

4.3.2.2 Component Persistence and Behavior

Chapter 3.2.2 of the "full" CCM specification is not retained in this profile because it refers to persistence that is not in the scope of this profile.

4.3.2.3 Implementing a CORBA Component

The whole chapter is retained.

4.3.2.4 Behavioral elements: Executors

The whole chapter is retained.

4.3.2.5 Unit of Implementation: Composition

In the CCM specification, the unit of implementation is called the *composition* referring to the keyword **composition** used in the CIDL to describe the whole component.

The term of composition is re-used here to be consistent with to the original specification but it does not rely on CIDL concept. The unit of implementation for lightweight CCM profile is detailed below:

- The ***component home type*** is described in IDL3.
- The ***abstract storage home binding*** is excluded from the specification as long as persistence notions are not retained.

- ***Home executor*** is reduced here to the most simple form, it means a factory to create components of a given type.
- ***Component executor*** is unique and does not support segmentation.
- The specification for home ***operation delegation*** is not retained because homes are reduced to simple component factories.
- ***Proxy home*** feature is not a necessary element for the kind of applications targeted by this profile. No needs are identified to relocate home operations.

4.3.2.6 Composition Structure

The composition for lightweight CCM is limited to a ***home type*** and the ***life cycle category*** of the component implementation**.**. For the life cycle category, see the chapter 4.3.2.1.The other information needed for the code generation is the ***home_executor_name*** and the ***executor_name.*** They can be automatically generated using the *component type* and the *home type* available in the IDL3 definition.

Hereafter is the OMG CCM specification example 1 restated. It underlines the potential downsizing of the CCM. This example is based on the use of a "*Session"* container as it is in the "full" CCM specification.

```
-----------------------------------------------------------------------------------
// Example 1
//
// USER-SPECIFIED IDL
//
module LooneyToons {
        interface Bird {
                void fly (in long how_long);
        };
        interface Cat {
                void eat (in Bird lunch);
        };

        component Toon {
                provides Bird tweety;
                provides Cat sylvester;
        };
        home ToonTown manages Toon {};
};
```

```
// Example 1
//
// USER-SPECIFIED CIDL
//
import ::LooneyToons;
module MerryMelodies {
        // this is the composition:
        composition session Toon {
                home executor ToonTown {
                        implements LooneyToons::ToonTown;
                        manages ToonSessionImpl;
                };
        };
};
```

The CIDL code example is not needed.
The lightweight CCM framework will generate the following artifacts:
• The skeleton for the component executor ***ToonSessionImpl***

*('**Impl'** is a possible automatic suffix for a component executor. This suffix could be specified in a configuration file of the framework or imposed)*

*• The complete implementation of the home executor **ToonTownImpl***
*('**Impl'** is a possible automatic suffix for a component executor. This suffix could be specified in configuration file of the framework or imposed).*

```
-------------------------------------------------
// Example 1
//
// GENERATED FROM IDL SPECIFICATION:
//
package LooneyToons;
import org.omg.Components.*;
public interface BirdOperations {
        public void fly (long how_long);
}
public interface CatOperations {
        void eat(LooneyToons.Bird lunch);
}
public interface ToonOperations extends CCMObjectOperations {
        LooneyToons.Bird provide_tweety();
        LooneyToons.Cat provide_sylvester();
}

public interface ToonTownExplicitOperations
extends CCMHomeOperations { }

public interface ToonTownImplicitOperations extends KeylessCCMHomeOperations {
        Toon create();
}

public interface ToonTownOperations extends
        ToonTownExplicitOperations,
        ToonTownImplicitOperations {}
-------------------------------------------------
```

The operations provide_tweety and provide_sylvester are excluded because only generic ones are retained.
The ToonImpl executor skeleton class has the following form:.

```
-------------------------------------------------
// Example 1
//
// GENERATED FROM IDL3 SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;
import org.omg.Components.*;
abstract public class ToonSessionImpl
implements ToonOperations, SessionComponent,
ExecutorSegmentBase
{
        // Generated implementations of operations
        // inherited from SessionComponent is omitted here.
        //
        protected ToonSessionImpl() {
                // generated implementation ...
        }
        // The following operations must be implemented
        // by the component developer:
        abstract public BirdOperations
        _get_facet_tweety();
        abstract public CatOperations
        _get_facet_sylvester();
}
-------------------------------------------------
```

The generated executor abstract base class ToonSessionImpl *implements all of the operations inherited by ToonOperations, including operations on* CCMObject *and its base interfaces. It also implements all of the*

operations inherited through SessionComponent, which are internal operations invoked by the container and the internals of the home implementation to manage executor instance lifecycle.
A complete implementation of the home executor ToonTownImpl *is generated from the IDL3 and descriptor XML files specification:*

```
-------------------------------------------------
// Example 1
//
// GENERATED FROM IDL3 SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;
import org.omg.Components.*;
public class ToonTownImpl
implements LooneyToons,ToonTownOperations,
HomeExecutorBase, CCMHome
{
        // Implementations of operations inherited
        // from ExecutorBase and CCMHome
        // are omitted here.
        //
        // ToonHomeImpl also provides implementations
        // of operations inherited from the component
        // home interface ToonTown
        CCMObject create_component()
        {
                return create();
        }
        void remove_component(CCMObject comp)
        {
        }
        Toon create()
        {
        }
        // and so on...
}
-------------------------------------------------The
user-provided executor implementation must supply the following:
• Implementations of the operations _get_tweety and
_get_sylvester, which must return implementations of the
BirdOperations and CatOperations interfaces
• said implementations of the behaviors of the facets tweety and
sylvester, respectively
The following example shows one possible implementation strategy:
-------------------------------------------------
// Example 1
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;
public class myToonImpl extends ToonSessionImpl
implements BirdOperations, CatOperations {
        protected long timeFlown;
        protected Bird lastBirdEaten;
        public myToonImpl() {
                super();
                timeFlown = 0;
                lastBirdEaten = nil;
        }
        public void fly (long how_long) {
                timeFlown += how_long);
        }
        public void eat (Bird lunch) {
                lastBirdEaten = lunch;
        }
        public BirdOperations _get_facet_tweety() {
```

```
        return (BirdOperations) this;
        }
        public CatOperations _get_facet_sylvester() {
        return (CatOperations) this;
        }
}
```

This simple example implements all of the facets directly on the executor. This is not the only option; the programming objects that implement BirdOperations *and* CatOperations *could be constructed separately and managed by the executor class.*
The final bit of implementation that the component programmer must provide is an extension of the home executor that acts as a component executor factory, by implementing the create_executor_segment *method. This class must also provide an implementation of a static method called* create_home_executor *that returns a new instance of the home executor (as an ExecutorSegmentBase). This static method acts as an entry point for the entire composition.*

```
// Example 1
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;
public class myToonTownImpl extends ToonTownImpl
{
        protected myToonTownImpl() { super(); }
        ExecutorSegmentBase
        create_executor_segment (int segid) {
        return new myToonImpl();
        }
        public static ExecutorSegmentBase
        create_home_executor() {
        return new myToonTownImpl();
        }
}
```

Note that these last two classes constitute the entirety of the code that must be supplied by the programmer. The implementations of operations for navigation, object reference creation and management, and other mechanical functions are either generated or supplied by the container.

4.3.2.7 Compositions with Managed Storage

Chapter 3.2.7 of the “full” CCM specification is not retained in this profile because it refers to persistence that is not in the scope of this profile.

4.3.2.8 Relationship between Home Executor and Abstract Storage Home

Chapter 3.2.8 of the “full” CCM specification is not retained in this profile because it refers to persistence that is not in the scope of this profile.

4.3.2.9 Executor Definition

The only element retained for the executor definition is the name of the executor, and it should be automatically generated according to the type of the element (see Example 1).

Chapter 3.2.9.1 of the "full" CCM specification is not retained in this profile because it refers to *segmented executors* which are not in the scope of this profile.

Chapter 3.2.9.1 of the "full" CCM specification is not retained in this profile because it refers to *delegation of feature state* which is not in the scope of this profile.

4.3.2.10 Proxy home

Chapter 3.2.10 of the "full" CCM specification is not retained in this profile because it refers to *proxy home* which is not in the scope of this profile.

4.3.2.11 Component Object References

The introduction of this chapter is retained, except the paragraph concerning the **Entity2Context** because *Entity* container API type is not retained.

Only sub-chapters "*Facet Identifiers*" (3.2.11.1), "*Monolithic Reference Information*" (3.2.11.4) and "*Component identity*" (3.2.11.6) are retained. The others refer to *persistence* and *segmentation* of component and are not in the scope of this profile.

4.3.3 Language Mapping

The following points expose what is relevant for this profile:

- Only *monolithic* strategy is retained in this profile because, as it is mentioned in the CCM specification, it is the most simple and it is sufficient for most use cases. It means that all sections concerning *locator* strategy will be excluded. Only **HomeExecutorBase** is retained.
- The notion of internal interfaces (provided by the container for the component) and callbacks (provided by the component for the container) is a convenient feature for applications targeted.
- The context reference of the component and associated **Context** object is suitable in our case because it permits interactions between container and component.

4.3.3.1 Common Interfaces

The **EnterpriseComponent** callback interface is retained.

The **ExecutorLocator** is not retained because segmentation of components is not retained..

4.3.3.2 Mapping Rules

4.3.3.2.1 Interfaces and Eventtypes

Mapping rules for ***Interfaces, Eventtypes*** are retained.

4.3.3.2.2 Components

For ***Components***, only the *monolithic exexutor* callback and the *context* internal interface are retained. This profile only treats the case of *service* and *session* components which implies the inheritance of **Components::SessionComponent** for the implementation of a *monolithic executor*, and the inheritance of **Components::SessionContext** for the implementation of the container.

4.3.3.2.3 Example

Here is the example of the "full" CCM specification, modified for this profile:

For the following component declaration in IDL :

```
interface Hello {
        void sayHello ();
};
component HelloWorld supports Hello {
        attribute string message;
};
```

the following local interfaces are generated:

```
local interface CCM_Hello : Hello
{
} ;

local interface CCM_HelloWorld_Executor :
Components::EnterpriseComponent, Hello
{
        attribute string message;
};

local interface CCM_HelloWorld :
Components::EnterpriseComponent, Hello
{
        attribute string message;
};

local interface CCM_HelloWorld_Context :
Components::CCMContext
{

};
```

4.3.3.2.4 Ports

All the operations redundant with the generic ones are excluded from this profile. They were the following:

CCM_<type> get_<name>()for facets.
<type> get_connection<name>() and **<name>Connections get_connections_<name>()** for receptacles.
void push_<name>(in <type>ev) for publisher and emitter.

All the operations relating to the locator strategy are excluded.

4.3.3.2.5 Homes

Home Explicit Executor Interface is retained only for factory operations and doesn't support inheritance and supported interfaces (see chapter 4.1.7).

Home Implicit Executor Interface is retained only for keyless homes.

Home Main Executor Interface is retained:

```
local interface CCM_<home name> :
        CCM_<home name>Explicit,
        CCM_<home name>Implicit
        {
        };
```

Factories: language mapping is the same in this profile.

Finders are not in the scope of this profile.

Entry Points are not part of the language mapping; they are dealt with in the new OMG specification "*Deployment and Configuration*" (*mars/2003-03-04)*

Example:

```
home Bank manages Account {
        factory open (in string name);
        void close (in string name);
};
```

In this example, the following equivalent interfaces would be generated :

```
local interface CCM_BankExplicit :
Components::HomeExecutorBase
{
        Components::EnterpriseComponent open (in string name);
        void close (in string name);
};

local interface CCM_BankImplicit :
{
        Components::EnterpriseComponent create ()
        raises (Components::CCMException);
};

local interface CCM_Bank :
        CCM_BankExplicit,
        CCM_BankImplicit
{
};
```

The user would then implement the **CCM_Bank** interface and eventually provide an entry point that creates a **CCM_Bank** instance.

4.4 The Container Programming Model - CCM Chapter 4

4.4.1 Introduction

In general, all references to CIDL to describe features of a component, a home or a container are now references to IDL3 and possibly to XML files.

The overall architecture depicted in Figure 4-1 of the "full" CCM specification is not modified.

Comments about EJB are excluded from this profile.

- In this profile, homes with primary keys are excluded.
- Only the **stateless** and **conversational** usage model are retained.
- The only container API type retained is the **session** one because this profile only treats components using transient object references.
- It implies the exclusion of two component categories which are **Process** and **Entity. S**o **Session** and **Service** components are retained.

4.4.2 The Server Programming Environment

4.4.2.1 Component Containers

Internal interfaces and *Callback interfaces* are retained in this profile.

4.4.2.2 CORBA Usage Model

For component references, only **TRANSIENT** objects are retained in this profile and they don't support the *finder design pattern*.

Servant to object id mapping which can be used is **1:1** or **1:N** because only **stateless** and **conversational** CORBA usage models are retained.

Concerning threading policies, because CIDL is not retained, the definition of the chosen threading policy (**serialize** or **multithread**) must be done in an XML file used to generate the container implementation (like **.ccd** files specified in the "full" CCM specification).

4.4.2.3 Component Factories

The whole chapter is retained.

4.4.2.4 Component Activation

The whole chapter is retained.

4.4.2.5 Servant Lifetime Management

In this profile, only the **component, method** and **container** servant life time are retained because **transaction** is not in the scope of this profile. These

policies are defined in the "*CORBA Component Descriptor*" XML file. (see chapters 6 and 7 of the "full" CCM specification)

4.4.2.6 Transactions

Chapter 4.2.6 of the "full" CCM specification is not retained in this profile because it refers to transactions which are not in the scope of this profile.

4.4.2.7 Security

Chapter 4.2.7 of the "full" CCM specification is not retained in this profile because it refers to security aspects which are not in the scope of this profile.

4.4.2.8 Events

Lightweight CCM events are based on the OMG "*Lightweight Event Service*".

Attributes described in this chapter are the same for the lightweight CCM except the notion of filter which is not supported by "*Lightweight Event Service*".

The container is responsible for mapping events operations to the CORBA lightweight event service.

"*Transaction policies for Events*" chapter is not in the scope of this profile.

4.4.2.9 Persistence

Chapter 4.2.9 of the "full" CCM specification is not retained in this profile because it refers to persistence aspects which are not in the scope of this profile.

4.4.2.10 Application Operation Invocation

This chapter is retained excepting transaction aspects.

4.4.2.11 Component Implementations

A component implementation only consists here of one executor which describes the implementation characteristics of a particular component. It corresponds to the session container type.

4.4.2.12 Component Levels

Basic and **extended** (without persistence and multi-segment aspects) component levels are retained.

4.4.2.13 Component Categories

Service component is implicitly retained because **stateless** CORBA usage model and **session** container API type are retained.

Session component is implicitly retained because **conversational** CORBA usage model and **session** container API type are retained.

Process component is implicitly excluded because **durable** CORBA usage model and **entity** container API type are not retained.

Entity component is implicitly excluded because **durable** CORBA usage model, **entity** container API type and **PrimaryKey** are not retained.

4.4.3 Server Programming Interfaces – Basic Components

4.4.3.1 Component Interfaces

Internal, External and **Callback** interface are retained for this profile.

CCMContext and **EnterpriseComponent** are retained.

UserTransaction interface is excluded because transactions are not treated in this profile.

4.4.3.2 Interfaces Common to both Container API Types

Because security and transactions are excluded, the **CCMContext** internal interface for Lightweight CCM is as follows:

```
typedef SecurityLevel2::Credentials Principal; exception IllegalState { };
local interface CCMContext {
        Principal get_caller_principal();
        CCMHome get_CCM_home();
        boolean get_rollback_only() raises (IllegalState);
        Transaction::UserTransaction get_user_transaction()
        raises (IllegalState);
        boolean is_caller_in_role (in string role);
        void set_rollback_only() raises (IllegalState);
};
```

Only **get_CCM_home** is retained because other operations deal with transactions and security.

UserTransaction interface is not retained.

There is no change on the **EnterpriseComponent** interface.

4.4.3.3 Interface Supported by the Session Container API Type

The whole **SessionContext** interface is retained.

The whole **SessionComponent** interface is retained.

The **SessionSynchronisation** interface is excluded from this profile because it deals with transactions.

4.4.3.4 Interface Supported by the Entity Container API Type

Chapter 4.3.4 of the "full" CCM specification is not retained in this profile because it refers to persistence aspects which are not in the scope of this profile.

4.4.4 Server Programming Interfaces – Extended Components

4.4.4.1 Interfaces Common to both Container API Types

The **CCM2Context** interface is retained but the only operation needed is the **req_passivate**. Other operations provided are not essential in this profile since **HomeFinder** are not used, and persistence and transaction are not supported. Note that the **get_event()** operation will be deprecated in the next release of the CCM specification.

The **CCM2Context** for lightweight CCM is as follows:

```
~~typedef CosPersistentState::CatalogBase CatalogBase;~~
~~typedef CosPersistentState::TypeId TypeId;~~

exception PolicyMismatch { };
~~exception PersistenceNotAvailable { };~~

local interface CCM2Context : CCMContext {
        ~~HomeRegistration get_home_registration ();~~
        ~~EventsNotification::Event get_event();~~

        void req_passivate () raises (PolicyMismatch);
        ~~CatalogBase get_persistence (in TypeId catalog_type_id)~~
        ~~raises (PersistenceNotAvailable);~~
};
```

The **HomeRegistration** and **ProxyHomeRegistration** interfaces are excluded because **HomeFinders** are excluded.

4.4.4.2 Interfaces Supported by the Session Container API Type

The **Session2Context** interface is not useful for the targeted environment where components don't need to be self managed in this way. So the whole chapter is excluded from this profile.

4.4.4.3 Interfaces Supported by the Entity Container API Type

Chapter 4.4.3 of the "full" CCM specification is not retained in this profile because it refers to persistence aspects which are not in the scope of this profile.

4.4.5 The Client Programming Model

The two forms of clients are supported in this profile. From the client's perspective, the home supports only one design patterns - factories for creating new objects. There is no **primarykey** parameter in the home IDL3.

4.4.5.1 Component-aware Clients

In this profile, component-aware clients locates CORBA interfaces of the component unsing a naming service.

- Initial references used by the component client are as follow for the lightweight CCM:
  - Name Service ("**NameService**")
  - ~~Transaction Current ("**TransactionCurrent**")~~
  - ~~Security Current ("**SecurityCurrent**")~~
  - ~~Notification Service ("**NotificationService**")~~
  - ~~Interface Repository ("**InterfaceRepository**") for DII clients~~
  - ~~Home Finder ("**ComponentHomeFinder**")~~
  - Event Service ("**EventService**") is added in this profile
- Concerning *Factory Design Pattern* section, since this profile doesn't support the **HomeFinder**, the code fragment presented in the "full" CCM specification is modified as follow:

```
// Resolve NamingService
org.omg.CORBA.Object objref =
orb.resolve_initial_reference("NamingService");
NamingContext ncRef = NamingContextHelper.narrow(objref);

// Resolve the Home Object reference in Naming
NameComponent nc = new NameComponent("AHome", "");
NameComponent path[] = {nc};
AHome homeRef = AHomeHelper.narrow(ncRef.resolve(path));

org.omg.Components.CCMObject AInst=
homeRef.create_component();

A Areal = AHelper.narrow(AInst);

// Invoke Application Operation
answer = A.foo(input);
```

- *Finder Design Pattern* section is excluded because *finder* operations are not treated in this profile.
- *Transactions* and *Security* sections are excluded.
- *Events* section is retained but deals now with "*Lightweight Event Service*" instead of notification service.

4.4.5.2 Component-unaware Clients

In that case, only supported interfaces can be used by the client.

- Techniques described for the *Factory Design Pattern* can be also used in this profile if **CosLifeCycle::FactoryFinder** are available in low foot-print environment.
- *Finder Design Pattern* section is excluded.

- *Transactions* and *Security* sections are excluded.
- *Events* section is retained but deals now with "*Lightweight Event Service*" instead of notification service.

4.5 Integrating with Enterprise JavaBeans - CCM Chapter 5

This chapter is not in the scope of this profile. JavaBeans are not required for embedded targeted environment.

4.6 Packaging and Deployment - CCM Chapter 6

For the lightweight CCM, this section of the specification will be based on the OMG "*Deployment and Configuration*" specification (*mars/2003-03-04*).

4.7 XML DTDs - CCM Chapter 7

For the lightweight CCM, this section of the specification will be based on the OMG "*Deployment and Configuration*" specification (*mars/2003-03-04*).