CORBA Component Model: *Discussion and Use with OpenCCM*

Raphaël Marvie, Philippe Merle

Laboratoire d'Informatique Fondamentale de Lille
UPRESA CNRS 8022
Bât. M3 – UFR d'I.E.E.A.
F-59655 Villeneuve d'Ascq (France)
{marvie,merle}@lifl.fr

Keywords: CORBA Component Model, Distributed Component Based Applications, Middleware, OpenCCM

Edited by:

Received: **Revised:** **Accepted:**

Distributed Object Computing (DOC) middleware has introduced the use of Interface Definition Languages to improve the design, production, and execution of large scaled critical distributed applications. Nowadays, software engineering is evolving to Distributed Component Computing (DCC) middleware in order to also address deployment and administration of such applications. In this context, the Object Management Group proposes the CORBA Component Model (CCM) as "the first industrial heterogeneous model". This model seems to be the proposal with the highest potential, compared to Sun Microsystems' Enterprise Java Beans and Microsoft's Distributed Component Model, to design, produce, deploy, and run distributed heterogeneous component based applications. Nevertheless, no implementation covers it in its whole. This article aims at discussing the CCM and to introduce OpenCCM—the first available open source implementation of the CCM. In that, most of the CCM concepts are illustrated using an example developed with OpenCCM.

1 Introduction

Large scaled critical distributed applications, such as telecommunication ones, require the use of technical models to ease their production and environments to improve their reliability. Nowadays, most complex distributed applications are built upon Distributed Object Computing (DOC) middleware like the *Common Object Request Broker Architecture* (CORBA) [22] of the Object Management Group (OMG), the *Distributed Component Object Model* (DCOM) [9] of Microsoft, or the *Java Remote Method Invocation* (RMI) [18, 30] of Sun Microsystems. This is because such middleware has demonstrated their ability to rationalize and to improve the software production process of distributed applications which includes design, production, and execution steps. The major contribution is the systematic use of Interface Definition Languages (IDL) to define object contracts and to automate the generation of code related to communications (stubs and skeletons) [10]. In that, type checking of remote invocations is performed at compilation time to produce error proof applications. Moreover, an application is seen as a set of interconnected distributed objects. Similarly, system services like naming, trading, notification, security, transaction, and persistence are also distributed objects, thus offering a homogeneous vision of the distributed system.

Unfortunately, such middleware includes several drawbacks that reduce the correct support of configurable and large scaled distributed applications. First of all, connections between objects are hidden deep inside their implementations and cannot be easily configured by architects from the outside. Then, it is hard to have a proper vision of a distributed application architecture [1, 2, 5]. Second, the use of system services, the non-functional aspects of an application, has to be hard-coded in the body of objects, thus mixed with their functional part. This leads developments to be more complex for business domain experts: They have to master both the business logic of their applications and the technical complexity of using system services. Finally, the deployment phasis of objects is not taken into account. There is no standard paradigm to install objects on their execution sites automatically: There are only *ad hoc* solutions and tools.

In the meantime, academic and industrial projects have shown that the component paradigm was matching software engineering requirements in a better way than the object paradigm [3, 7, 16, 25]. Thus, to bypass above limits, middleware has evolved from an object based to a component based approach like *Enterprise Java Beans* (EJB) [15] of Sun Microsystems and the more recent *CORBA Component Model* (CCM) [20] of the OMG. Such middleware promotes the use of container servers to host component instances and to implicitly take into account system services. Thus, component implementations can, at least, contain only their business logic. Moreover, strategies for the use of required system services are defined in descriptors that are external from the implementations. Then, it is possible to apply different strategies on a single component implementation, providing distinct behavior without modifying the business code. For instance, EJB and CCM specifications define *eXtended Markup Language* (XML) grammars (*i.e. Data Type Definitions*) to describe component requirements in terms of life cycle, persistence, transactions, and security. Then, component packages—type, implementation, and descriptors—are deployed in container servers. Containers are configured according to the descriptors to properly provide system services to component instances. The mentioned architectural drawback is partially over pass: EJB components may be interconnected using the listener design pattern defined in the *Java Beans* [6] and the CCM introduces the concept of *port* as discussed in Section 3. Thanks to Distributed Component Computing (DCC) middleware, applications are now viewed as a composition of packaged, deployable, configurable, and distributed components.

In the context of this evolution, the CORBA Component Model tends to have the highest potentialities to globally cover the process of designing, developing, packaging, deploying, and executing distributed heterogeneous components. Research work is already in-progress in various academic projects [12, 13, 27, 28, 29] to evaluate, implement, and extend the CCM and its use. Nevertheless, the specification is in its last step before standardization and no complete implementation of the CCM is available. OpenCCM [14], developed in our research team at the LIFL, is the most advanced CCM platform available at the moment[1]. This article presents the specification close to its first release and relies upon OpenCCM to illustrate the discussion.

The CCM specification defines five models and a meta-model. As the CORBA Component abstract model is the most finalized part of the CCM specification, this article will entirely cover this model. Then, other models are outlined and their use is illustrated with examples written using OpenCCM. In that, models are mainly presented according to their implementation in OpenCCM. Section 2 presents an overview of the CCM and of the OpenCCM platform. Section 3 discusses the CORBA Component abstract model using an example to present how component types are defined. Section 4 presents both the programming model of the CCM and its use with OpenCCM. Section 5 briefly outlines the definition of component package formats. Section 6 discusses the deployment model and section 7 the runtime model as well as the available means of OpenCCM regarding these models. Finally, section 8 concludes this article summarizing the good points and drawbacks of the CCM specification, a comparison of EJB and CCM, as well as OpenCCM perspectives.

2 CCM and OpenCCM Overview

2.1 Which Kind of Applications? Which Concepts?

EJB based applications are made of homogeneous components written in Java only and deployed on

[1] A C++ platform is also available upon MICO.

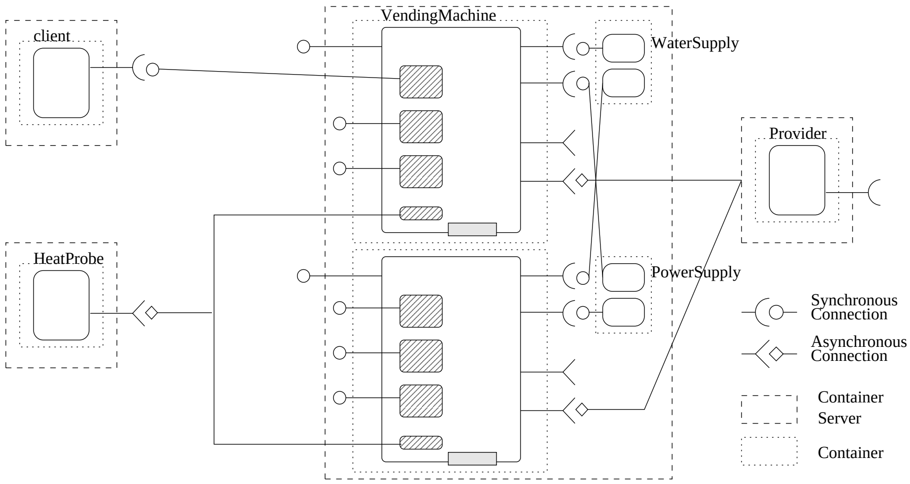

Figure 1: CCM Based Distributed Application

a single host, the application server. DCOM based applications could be made of heterogeneous components but written for a single Operating System, MS Windows. CORBA component-based applications are intended to be built using heterogeneous distributed components. Moreover, components of a single application may be developed using various programming languages and for several Operating Systems, and deployed in several servers of a distributed system. Figure 1 depicts an overview of what might be a CORBA component-based application.

This application is a possible instantiation of the *toy* application vending machine that is used to illustrate this article. This application instance runs on four different hosts that are remote from one another. A central site hosts two instances of vending machine components, which is common in most firm corridors, one for cold drinks and another one for warm drinks. In order for those two instances to run, an execution support is required: It is provided by a container. A container is a generic execution support that may host different kinds of component types that share common system requirements, like life cycle, persistence, transaction, and security needs.

In order not to reduce execution site capabilities to a fixed set of statically defined containers, and in order not to have to instantiate all the potential containers on a site, containers rely on container servers. These servers are processes, in the Unix meaning, that permit one to instantiate containers on demand and according to specific system requirements of applications. Inside the main container server, there are four containers that allow the execution of six component instances: Two vending machine instances, two power plug instances, and two water supply instances.

Given that a container has just been defined as a generic execution environment and in order not to limit the behavior of component instantiation, a *component home* is used. A home is an instance manager that takes care of component instance life cycle. A home is to component instances what the *new* operator is to objects. Component homes contain the code reifying component instantiation, which brings flexibility. They are equivalent to the ‘static’ part of an object’s class. Component homes are designed according to the factory design pattern [8]. In addition of permitting the use of the same container with different component types, this approach also brings automation in the deployment of applications.

Thus, it is possible to choose execution sites from a known set, and then to install an application on these hosts from an administration site. To do so, in addition of providing an execution environment (the container server) a host has to provide a facility to download code. It is then possible to upload a component implementation on a remote host, as well as its associated home implementation. Once containers are instantiated, the implementation of components and their home uploaded, home and component instantiated, connecting component instances together is the last step to build the application.

Connecting two component instances is performed using *ports*, the connection points. In order for two component types to be connectable, it is required that both share a common knowledge: A port provided by one of them has to be compatible with a port used by the other. Figure 1 depicts the two connection modes between component types supported by the CCM: A synchronous one, using method invocation, and an asynchronous one, through event notification.

2.2 CCM Overview

The CCM defines a server side component framework, but it may also be used on the client side of applications as illustrated in the vending machine example. The CCM is split in two distinct levels: A basic level that is useful to *componentize* classic CORBA objects, and a richer and more interesting extended level that is presented in this article. Like EJB, to which the basic level using the Java language is equivalent, the CCM technology relies on the use of containers to host component instances and to ease their deployment. The CCM specification [20], which is more than one thousand pages, is structured in five models and a meta-model.

The abstract model provides application designers a means to define the various interfaces provided and used by a component type, as well as its properties. In that, the OMG Interface Definition Language has been extended to take into account the new concepts introduced in the CCM, especially the definition of *ports* which is the means to define component type with multiple interfaces. The abstract model also allows designers to define managers for component instances.

The programming model includes a framework, the *Component Implementation Framework* (CIF) that defines how functional (*i.e.* the business logic that is programmed) and non-functional (*i.e.* generated automatically) have to cooperate. This framework relies upon the use of the *Component Implementation Definition Language* (CIDL). It also defines the way component instances could interact with the container hosting them.

The packaging model specifies how component types and implementations should be packaged. These component packages are then usable to sell and deploy components over networks. It is associated with a set of descriptors written in the *Open Software Description* (OSD) language, an eXtended Markup Language (XML) grammar (*i.e.* DTD).

The deployment model defines a process that permits one to deploy an application over several execution sites in a simple and automated way. This process uses component packages to install component implementations, create component instances, and to interconnect them together in order to provide a running application.

The execution model describes the execution environment of component instances. The main role of containers is to hide and to manage non-functional aspects of a component, like life cycle and persistence, such as it is no more necessary to program the use of associated CORBA services.

The meta model defines the concepts of the various models mentioned here, as well as their relationships. This meta model is defined using UML schemas and the MOF as context. For example, the OMG IDL meta model includes the definition of the `component` meta type as well as the various new concepts such as port (see Section 3).

2.3 From the Design to the Deployment of CCM Applications

Figure 2 depicts the whole development and packaging process of CORBA component based applications. It underlines the structuration of the software production process in several steps that are addressed by the CCM, providing them a set of means. These means allow each actor of this process to cooperate with the others in order to have a coherent multi-actor approach. This production cycle of CORBA component-based applications is discussed all along this article, presenting the various means provided to address it.

A designer defines (1) the component types using the new OMG IDL, that we will name

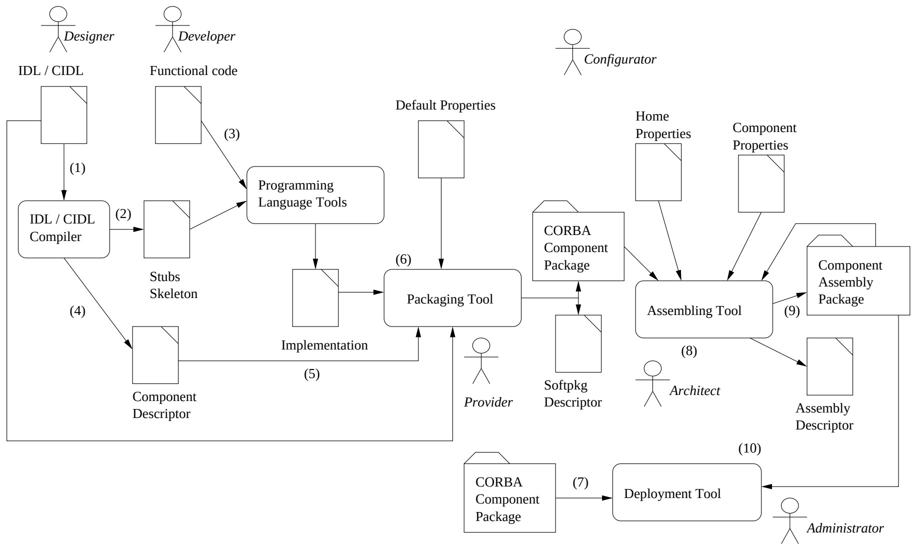

Figure 2: Global CCM Software Production Process

OMG IDL3 from now. These definitions are then mapped (2) in the CORBA 2 OMG IDL, that we will named OMG IDL2 from now, in order to allow the implementation (3) of these component types by developers. The mapping step also produces (4) a component descriptor template, that has to be finalized (5) by the developer. Once implemented, a component type is packaged (6) into an archive with its descriptor, a default configuration, and one or several implementations (*e.g.* in Java and C++) in order to be spread out. This package may be later on deployed *as is* (7) or included (8) by an architect in component assemblies in order to build large applications. To be diffusible, a component assembly is also packaged (9), grouping into an archive the assembly descriptor, the component packages or their location, and a default configuration of each component for this specific context. Assembly packages are then spread out to be used *as is* as well as for future composition. Finally, packages of components and assemblies are deployed to instantiate applications on execution sites (10).

2.4 OpenCCM Platform

The CCM is specified, however there are few implementations. Moreover, very few using experiences are reported in litterature. OpenCCM was the first implementation publicly released. At first, the objective of our work was not to implement the whole specification of the CCM, but only the parts that were required to support our research experiments. We have focussed our efforts on the abstract and deployment models. Our first goals were to have the following parts:

- the interface repository for the OMG IDL3, a first version is already available,
- an OMG IDL3 compiler to load OMG IDL3 files into the repository, maps it to OMG IDL2, as well as to generate component extended skeletons for the Java language (and C++ later on), a first version is already available.
- a CORBA component packaging tool (in progress),
- a model and its associated framework to define component based applications (in progress: CODeX—*Composite Oriented Deployment and eXecution*),
- a flexible deployment machine that supports the CCM deployment process, but that can also extend this process to bet fit a larger set of requirements, and
- the definition of adaptive containers that would look like composites, allowing to compose and deploy containers according to application requirements (in progress).

Finally, as we are now member of the CCM Finalization Task Force at the OMG, that we want the specification to be adopted (which requires at least one implementation), and that we are please to contribute, our motivation has evolved to producing an industrial quality open source CCM platform: OpenCCM [14]. This platform is already available at `http://corbaweb.lifl.fr/OpenCCM`, over ORB such as ORBacus 4 for Java [23], OpenORB [24], and VisiBroker 4 for Java [11]. Every one is encourage to use it and to contribute.

3 Defining Component Types

The CORBA Component abstract model allows designers to define functional interfaces of components using the OMG IDL3 language. These definitions are only used for design purpose. To implement a component type, the OMG IDL3 definition is mapped into OMG IDL2. A mapping rule to OMG IDL2 is associated to each OMG IDL3 construction. It is from this OMG IDL2 that a developer will generate the implementation skeleton and to program the functional part of the component.

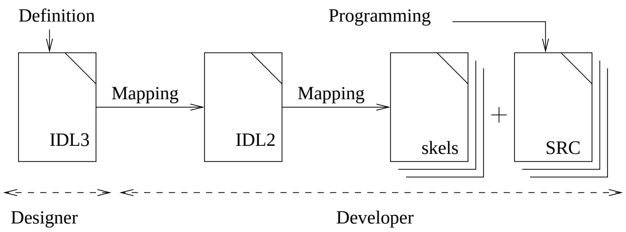

Figure 3: From OMG IDL3 to OMG IDL2

During this whole section, the steps depicted in Figure 3 will be used to present the various concepts of the abstract model. Each one is discussed like follows: The concept, its definition in OMG IDL3, the mapping rule to OMG IDL2, the meaning of this OMG IDL2 and its associated API, as well as a quick review. Finally, an review of the abstract model concludes this section.

3.1 Component Type

3.1.1 Concept

A component type groups attributes and *ports* definitions. Attributes are configurable properties of the component type, and are mainly intended to be used for configuration purpose. In the context of the CCM, attributes may raise exceptions when misused. *Ports* define interfaces (with the CORBA2 meaning) which are provided or used by the component type. Four kinds of *ports* are defined by the CCM (see Figure 4).

- **Facets** are interfaces provided by component types and synchronously used by clients. The vending machine component type provides three facets (`ClientFacet`, `ProviderFacet`, and `RepairmanFacet`) to be used.
- **Receptacles** are interfaces synchronously used by component types. The vending machine component type defines two receptacles (`PowerPlug` and `WaterPlug`) to be connected.
- **Event sinks** are interfaces provided by component types and asynchronously used by clients. The vending machine component type includes one event sink to consume `TemperatureEvt` events.
- **Event sources** are interfaces asynchronously used by component types. Two kinds of events are produced by the vending machine component type (`EmptyEvt` and `NoChangeEvt`).

These various ports will be used at deployment time and runtime to connect components together. For example, the `PowerPlug` facet provided by a power supply component will be connected to the `PowerPlug` receptacle of a vending machine component.

Like CORBA 2 objects, component instances are characterized by their interfaces and by a reference, named *base reference* in the context of the CCM. This reference allows clients to access ports of the component instance. Section 3.2 will show that other kinds of references exist in order to represent input ports.

3.1.2 OMG IDL3 Definition

Component types are defined using the `component` keyword (see Figure 5). This keyword represents

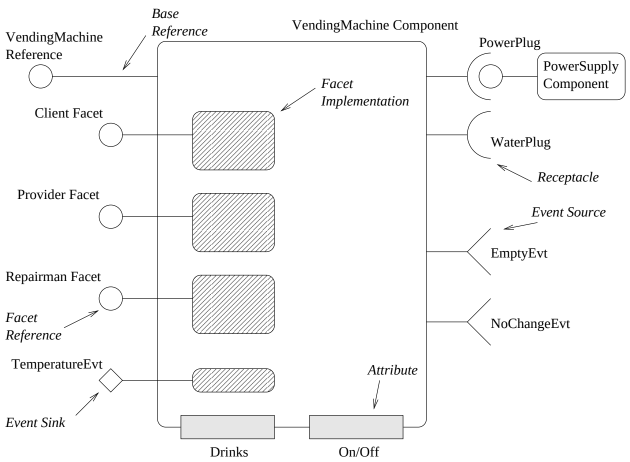

Figure 4: The CCM Abstract Model, Applied to the Vending Machine Example

a new meta-type which is a specialization of the interface one. Thus, component type definitions are very similar to interface definitions. Right now the vending machine component type includes the definitions of two attributes that will be configured according to each instance expected behavior. The two attributes are used to define which kind of drinks are available and if the machine is running or not.

```
component VendingMachine {
    attribute boolean on ;
    attribute DrinkSeq drinks ;
};
```

Figure 5: A CORBA Component Definition

Like interfaces, component types may be defined using a relation of inheritance. This inheritance relationship is *simple* “à la” Java: Component types could inherit from a single component type and may support one or several interfaces. Figure 6 depicts this approach: The `CardVendingMachine` component type offers the same attributes and ports as the standard `VendingMachine` one, as well as the `CardReader` operations. Then, it may also offer its own attributes and ports.

3.1.3 OMG IDL2 Mapping

An OMG IDL3 component definition is mapped to an OMG IDL2 interface definition. This interface contains operations mapping component ports

```
interface CardReader { ... };
component CardVendingMachine :
  VendingMachine supports CardReader {
  // ...
};
```

Figure 6: A CORBA Component Definition Using Inheritance

and inherits from the `Components::CCMObject` interface. This latter provides the common operations to all component types. Figure 7 depicts the mapping of previous component definitions. It is to be noticed that, once mapped, component inheritance is represented by CORBA 2 multiple inheritance.

```
interface VendingMachine :
  Components::CCMObject {

  attribute boolean on ;
  attribute DrinkSeq drinks ;
};

interface CardVendingMachine :
  VendingMachine, CardReader {
  // ...
};
```

Figure 7: OMG IDL2 Mapping of CORBA Component Type Definitions

Generated OMG IDL2 interfaces are used firstly by developers as implementation contracts of component types, and secondly by clients to interact with component instances at runtime.

3.1.4 `Components::CCMObject` Interface

`Components::CCMObject` is the base interface of any component type which is implicitly inherited. This interface inherits from the three interfaces `Components::Navigation`, `Components::Receptacles`, and `Components::Events` which are presented in the following sections. Operations defined in this interface allow clients to find back the home associated to a component instance (`get_ccm_home()`) as well as the primary key associated with the instance (`get_primary_key()`), to validate the configuration phasis (`configuration_complete()`) and to begin the execution phasis (see Section 3.6), as well as to destroy a component instance (`destroy()`). The `Components` module definitions related to the abstract model not depicted here may be found in the CCM specification.

```
module Components {
  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 destroy () ;
  };
};
```

Figure 8: `Components::CCMObject` Interface

3.1.5 Review

In the context of the CCM, attributes really represent configurable properties of component types. The exceptions associated to a misuse is also a benefit in terms of software quality. Then, being used to CORBA 2 philosophy, simple inheritance is surprising for component types. Like interfaces, a component type is a contract. There are no risks *a priori* of using multiple inheritance as it is not related to component implementations. Finally, as `component` is an extension of the `interface` meta-type, not being able to define operations unrelated to facets, and not using the `supports` construction, is strange. Moreover, the OMG IDL3 language is only a technical answer to the definition of component types. No methodology is prescribed for component type design. Such methodology has to be defined and used, OMG IDL3 definitions of component types being the result of this design step.

3.2 Facets

3.2.1 Concept

In the context of the CORBA 2 specification, an object has a unique OMG IDL interface grouping all the available operations. Dealing with inheritance, a reference may conform to several inherited interfaces, however the most derived type groups all the operations [29]. Thus, it is impossible to group operations of an object logically. Components overrun this limit.

A facet represents a point of view, or functional aspect, of a component type. Then, a facet may be defined for each role that can be performed upon the component. Looking at a vending machine, several interfaces are available: A consumer interface (with operations such as pay, select, take), a provider interface (with operations such as open, insert drinks, remove change, close), and an interface for repairmen (with operations such as open and close back door, check the pump). This argues the use of one facet per kind of users of the vending machine. Moreover, it avoids the consumer to think about opening the back door of the machine, which is a good basis to fix access control issues more easily.

A facet is specified by an OMG IDL interface. This interface is the unique available element to clients of the facet. Like objects, components rely upon the concept of encapsulation. In that, the implementation of a facet is hidden inside the implementation of the component. Thus, facets cannot be dissociated from the component they belong to. This implies for the life cycle of a facet to be related to the component one. In addition to the base reference of a component instance, each facet has its own reference (*facet reference*). Several copies of a facet reference may exist at a time, for example when multiple clients use the same facet of a component.

3.2.2 OMG IDL3 Definition

As depicted in Figure 9, the keyword `provides` is use to define a facet. The facet type has to be a valid OMG IDL interface. Such a definition arity is one, but it is possible to define multiple facets of the same type. Each facet has to be identified using a unique name. In our example, the vending machine component type provides three facets: One for clients of the vending machine, one for providers, and one for repairmen.

3.2.3 OMG IDL2 Mapping

Each OMG IDL3 facet definition is mapped to the definition of an operation which returns a reference of the facet type. This operation is named like the facet and prefixed with `provide_`. Then,

```
interface ClientFacet { ... } ;
interface ProviderFacet { ... } ;
interface RepairmanFacet { ... } ;
component VendingMachine {
    provides ClientFacet    client ;
    provides ProviderFacet  provider ;
    provides RepairmanFacet repairman ;
};
```

Figure 9: OMG IDL3 Facet Definition

for OMG IDL2 users, facets are accessible through a naming convention.

```
interface VendingMachine :
  Components::CCMObject {

  ClientFacet    provide_client () ;
  ProviderFacet  provide_provider () ;
  RepairmanFacet  provide_repairman () ;
};
```

Figure 10: Mapping to OMG IDL2 of Facet Definitions

3.2.4 Using Facets

The implementation of operations related to the component facets is part of the skeleton generated from OMG IDL2. This implementation is split in two levels: On one hand, there are the generic operations of the `Components::Navigation` interface, and on the other hand there are the operations specific to the component type that come from the OMG IDL3 to OMG IDL2 mapping. Generic operations could always be used and are useful for dynamic introspection. This use is flexible and powerful, but controlled at runtime only. Specific operations are more appropriate for static use. This use is fixed but reliable as controlled at compilation time. An interesting solution to implement specific operations is to rely on generic ones. This way, the name of the facet provided by a specific operation is correct, and then the generic operation will not be misused at runtime. In that, a developer only has the functional part of the facet implementation to write.

3.2.5 `Components::Navigation` Interface

In addition to offering several facets, a component provides a mean to navigate between them. Thus, a client can change its vision of the component during its execution, for example, the provider is thirsty. Navigation through facets is centralized around the base reference of a component. To permit navigation, a base interface is provided: `Components::CCMObject` inherits from `Components::Navigation`.

This latter provides generic operations available from and usable on any component type. This interface is implicitly inherited by any component type. Its main operations are introspection operations (`describe_facets()`) and facet retrieval ones (`provide_all_facets()`, `provide_named_facets()`, and `provide_facet()`). While looking for known facets, their names are used (name of the facet definition in the `provides` declaration of the component). Finally, an operation tests if two facets are part of the same component (`same_component()`).

```
module Components {
  interface Navigation {
    Object provide_facet
      (in FeatureName name)
      raises (InvalidName);
    FacetDescriptions describe_facets () ;
    Facets provide_all_facets () ;
    Facets provide_named_facets
      (in NameList names)
      raises (InvalidName) ;
    boolean same_component
      (in Object ref) ;
  };
};
```

Figure 11: `Components::Navigation` Interface

In the CCM context, any software entity becomes a component. The only use of the `interface` keyword is to define facets. Thus, it is important to be able to find back the component offering the facet for which one has a reference. The `CORBA::Object` interface now includes the `get_component()` operation, to find which component instance owns a given facet reference.

3.2.6 Review

Facets are an important benefit of the CCM specification. They avoid the definition of interfaces grouping 'in bulk' all the operations provided by a component. Now on, operations can be grouped according to the various component users kinds. Moreover, introspection facilities associated with facets permit one to discover the set of roles provided by a component type, as well as their use easily.

Nevertheless, the arity (one) and the co-location of facets can be seen as limits of the abstract model. In the meantime, facets lack the possibility of dynamic definition at runtime, they could only be defined statically at design time.

3.3 Receptacles

3.3.1 Concept

A receptacle allows a component type to accept a reference (*i.e.* a facet reference, a component reference, or just an object reference) in order to use the operations it provides. This relationship is named a connection. Then, a receptacle is a conceptual connection point. This abstraction is materialized in a component through a set of operations to set and manage connections.

A receptacle is used to build assemblies of components. This composition pattern allows one to build distributed applications more easily at deployment time. It also permits component instances to delegate processing requests. Receptacles may be of two distinct types: Simple receptacles (only one reference may be connected at a time), and multiple receptacles (several references may be connected simultaneously).

3.3.2 OMG IDL3 Definition

For both kinds, receptacles are defined using the keyword `uses` as depicted in Figure 12. If dealing with a multiple receptacle, the `multiple` keyword is associated with the `uses` keyword. Each receptacle is named using a unique identifier (unique to the component type definition).

In our example, the `WaterPlug` receptacle is simple, while the `PowerPlug` receptacle is multiple. The use of a simple receptacle means that there is at most one component instance connected at a time. Dealing with multiple receptacles, a limit for the maximum number of connections at a time may be defined in the component implementation.

```
interface PowerPlug { ... } ;
interface WaterPlug { ... } ;
component VendingMachine {
    uses WaterPlug          water ;
    uses multiple PowerPlug power ;
};
```

Figure 12: OMG IDL3 Receptacle Definitions

3.3.3 OMG IDL2 Mapping

The mapping of receptacle definitions is more complex than the mapping of facets. In the context of multiple receptacles, a structure is defined to manage multiple connections. For each connection, a *cookie* is created when established. This cookie identifies the connection uniquely and is used for any operation on this connection, like removing it. For both kinds of receptacles, connection and disconnection operations are defined. The mapping of the receptacles defined in Figure 12 is presented in Figure 13.

3.3.4 Using Receptacles

In order to use connections, the functional part of a component implementation is offered operations to be invoked on the receptacle to retrieve the reference connected to it. The way receptacles are managed may vary from a CCM platform to another, that is why developers have to use the `get_connection*()` operations. When asking connections using the receptacle name, a component implementation receives a sequence of structures containing the connection cookies as well as the connected references. In the context of a simple receptacle, only the connected reference is received.

Once a connection has been established between a receptacle and a reference, the component implementation could invoke operations using the reference at any time and without restrictions. The way the invocation will be processed only depends on the component implementation.

Connection The `connect*()` operations are partly generated and have partly to be imple-

```
interface VendingMachine : Components::CCMObject {
  void connect_water (in WaterPlug cnx)
    raises (Components::AlreadyConnected, Components::InvalidConnection) ;
  WaterPlug disconnect_water ()
    raises (Components::NoConnection) ;
  WaterPlug get_connection_water () ;

  struct powerConnection {
    PowerPlug         objref ;
    Components::Cookie  ck ;
  };
  typedef sequence<powerConnection> powerConnections;

  Components::Cookie connect_power (in PowerPlug cnx)
    raises (Components::ExceededConnectionLimit, Components::InvalidConnection) ;
  PowerPlug disconnect_power (in Components::Cookie ck)
    raises (Components::InvalidConnection) ;
  powerConnections get_connections_power () ;
};
```

Figure 13: Mapping to OMG IDL2 of Receptacle Definitions

mented by the component developer. In the context of multiple receptacles, the developer has to implement the management of the connected references. Once the connection operation has been performed, the receptacle keeps it until an explicit disconnection request is received.

Dealing with simple receptacles, only one connection may exist at a time, otherwise the `AlreadyConnected` exception is raised. Dealing with multiple receptacles, a limit may be set for the number of connections, when this number is reached, the `ExceededConnectionLimit` exception is raised while trying to establish a new connection. In both cases, the component implementation may refuse to establish a connection for an arbitrary reason, the `InvalidConnection` exception is then raised.

In the context of multiple receptacles, the `connect*()` operations return a cookie for the identification of the connection. This cookie is generated by the receptacle implementation and used by the client of the connection to act upon it[2]. Cookies are required as CORBA references could not be compared semantically.

Disconnection The `disconnect*()` operations remove the relationship between a component instance and the connected reference. In the context of a simple receptacle, the operation returns the reference that was connected to the receptacle, or the `NoConnection` exception is raised. In the context of a multiple receptacle, it is necessary to specify the cookie of the connection to end. If the cookie is not matching, the `InvalidConnection` exception is raised.

3.3.5 `Components::Receptacles` Interface

The `Components::Receptacles` interface provides generic operations to (dis)connect component instances together (see Figure 14). Operations are applied on the port which name is given as parameter. If operations are misused, like providing an unknown port name, exceptions are raised. The `get_connections()` operation provides the list of facets connected to a given receptacle.

3.3.6 Review

Receptacles are a means for composition purpose. Receptacles make components a real evolution compared to objects, as services used by the component are explicitly described while services used by objects are references deeply hidden in the object implementation. Then, it is possible to know which services / interfaces have to be provided to

[2]The cookie has to be kept by the client, as it cannot be retrieved a second time.

```
module Components {
  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) ;
    ConnectedDescriptions get_connections
      (in FeatureName name)
      raises (InvalidName);
  };
};
```

Figure 14: `Components::Receptacles` Interface

a component at runtime. Moreover, it is possible and quite simple to change connections during the life time of a component instance. This could be useful for reconfiguration purpose.

The semantic of multiple receptacles is not clearly defined in the CCM specification. A potential use is for fault tolerance purpose: Dealing with power supplies, if such a component instance does not work any more, then the vending machine component would use another one (which is already connected). Multiple receptacles could also be used to *callback* interfaces between a component type and others using its facets. This choice is dependent of the implementation. Then, it would be interesting to specify if all the connected facets have to receive invocations or if connected facets are used one at a time, and then how they are selected.

The lack of structural introspection of receptacles is a drawback right now. It is not possible to discover dynamically available receptacles on a component instance through an operation provided by its component type. Such introspection is possible through the Interface Repository, but an operation similar to `provide_all_facets()` (like `get_receptacles()`), would be a strong point regarding a dynamic and/or automatic use of component instances.

Finally, the `Components::Receptacles` interface seems to address more properly multiple receptacles than simple ones. The `get_connections()` operation seems, according to its name and its use (returned type), to be designed for multiple receptacles. Connection and disconnection operations are definitively designed for multiple receptacles according to the use of a cookie that does not exist when using simple receptacles. Nevertheless, these operations may be used to implement the mapping of simple receptacles. But, it looks like "adaptation" more than "being designed for".

3.4 Events

3.4.1 Concept

The event model is a classical producer / consumer one. This model is compatible with event and notification services of the CORBA Services specification [19]. To receive events, a client has to subscribe to the associated event sources. In the context of the CCM, only the mode *push* is used. Operations to process events are defined by consumer interfaces. Services to manage events are provided by the implementation of the containers. There are two kinds of ports that deal with events: Event sources and event sinks.

There are two kinds of event sources. An *Emitter* is an event source that works on a 1-to-1 basis, only one consumer may be connected at a time. In this context, there is no event channel between the source and the consumer, the connection is straight forward between the producer and the consumer. This connection is set up during the configuration of the component instance. A *Publisher* is an event source that works on a 1-to-n basis, many consumers may be connected to the source at the same time. In this context, the subscription of a consumer to an event type is delegated to an event channel provided by the container. The unique event source for such an event channel is the component instance for which the channel has been created.

An event sink permits a component instance to receive events of a given type. It is like a facet which type is an event consumer, inheriting the `Components::EventConsumerBase` base interface. On this side there is no difference between connection and subscription. The component instance does not control this consumer port. The instance only makes it public in order to receive events.

The event sink may then receive events from various sources in the same time.

3.4.2 OMG IDL3 Definition

The definition of event sources is based upon the use of the `emits` and `publishes` keywords. Event sinks are defined using the `consumes` keyword. Each port specifies the event type produced or consumed. The event type are `valuetypes` (*Object by Value*) inheriting from the `Components::EventBase` interface.

```
valuetype NoChangeEvt :
  Components::EventBase { ... } ;
valuetype EmptyEvt :
  Components::EventBase { ... } ;
valuetype TemperatureEvt :
  Components::EventBase { ... } ;
component VendingMachine {
  emits     NoChangeEvt     change ;
  publishes EmptyEvt        empty ;
  consumes  TemperatureEvt  temp ;
};
```

Figure 15: OMG IDL3 Definition of Two Event Sources and an Event Sink

Figure 15 depicts the definition of the vending machine event sources and sinks. The `change` port emits events related to the lack of change. Only one consumer may be connected at a time. The `empty` port publishes events to clients and providers to signal that there are no more drinks available. Finally, the `temp` port is an event sink for the machine to receive information regarding the evolution of the temperature.

3.4.3 OMG IDL2 Mapping

Like receptacle mapping rule, the mapping of events are quite complex as depicted in Figure 16. Dealing with event sources, an interface of event consumer (like `NoChangeEvt` consumer) is defined for clients of this source[3]. As stated before, events could only be pushed to consumers. This is why consumers provide a single operation: `push()`. Then, [dis]connection operations (`connect_change()` and `disconnect_change()`) as well as [un]subscription ones (`subscribe_empty()` and `unsubscribe_empty()`) are added to the interface of the component. All the operations generated during the mapping of the source definition of Figure 15 are depicted in Figure 16.

[3]The mapping presented here is the correct one, that may not be similar to the one depicted in the available specification.

```
interface NoChangeEvtConsumer :
  Components::EventConsumerBase {
  void push (in NoChangeEvt evt) ;
};
interface EmptyEvtConsumer :
  Components::EventConsumerBase {
  void push (in EmptyEvt evt) ;
};
interface VendingMachine :
  Components::CCMObject {

  void connect_change
    (in NoChangeEvtConsumer consumer)
    raises(Components::AlreadyConnected) ;
  NoChangeEvtConsumer disconnect_change ()
    raises(Components::NoConnection) ;
  Components::Cookie subscribe_empty
    (in EmptyEvtConsumer consumer)
    raises
    (Components::ExceededConnectionLimit) ;
  EmptyEvtConsumer unsubscribe_empty
    (in Components::Cookie ck)
    raises(Components::InvalidConnection) ;
};
```

Figure 16: OMG IDL Mapping of Event Sources (Emitter and Publisher)

The mapping of an event sink, see Figure 17, is very similar to the one for an event source. Nevertheless, the consumer interface (`TemperatureEvtConsumer`) is no more intended to be used by the client, but by the component itself. The `get_consumer_temp()` operation permits one to retrieve the event sink reference in order to connect it to an event source of another component instance.

3.4.4 Using Events

Two operations are generated by the OMG IDL3 compiler for an *emitter* definition. These operations permit one to connect and disconnect a consumer. When disconnecting a consumer, its reference is returned. The two operations generated for a *publisher* port are for subscription

```
interface TemperatureEvtConsumer :
  Components::EventConsumerBase {
  void push (in TemperatureEvt evt) ;
};
interface VendingMachine :
  Components::CCMObject {
  TemperatureEvtConsumer
  get_consumer_temp () ;
};
```

Figure 17: OMG IDL2 Mapping of an Event Sink

and removal of a consumer from the events generated by the port. The event source only publishes events in the dedicated event channel. When subscribing, a cookie is generated to identify the consumer subscription. This cookie is required to end a subscription. The operation available on an event sink returns a reference of the consumer that matches with the event type defined.

```
module Components {
  interface Events {
    EventConsumerBase get_consumer
      (in FeatureName sink_name)
      raises (InvalidName) ;
    Cookie subscribe
      (in FeatureName publisher_name,
       in EventConsumerBase subscriber)
      raises (InvalidName) ;
    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) ;
    EventConsumerBase disconnect_consumer
      (in FeatureName source_name)
      raises (InvalidName,
              NoConnection) ;
  };
};
```

Figure 18: `Components::Events` Interface

3.4.5 `Components::Events` Interface

The `Components::Events` interface (see Figure 18) provides a generic set of operations to access event sources and sinks of a component (`Components::CCMObject` inherits from `Components::Events`). The various operations permits one to [dis]connect and to [un]subscribe to an event source as well as to retrieve an event consumer.

3.4.6 Review

The mapping of events discussed here is, at time of writing, a correct version that works (unlike the first proposal in the specification). Now on, we claim that the use of a keyword to define event types is better than the use of inheritance. Thus, the use of the `eventtype` keyword should be standardized for the CCM finalization. The use of a keyword first formalizes a concept, and an event is a concept as well as component is one. Second, it is easier to define events and to use their definition for event sources and sinks. Then, the definition of an event should be written as depicted in Figure 19.

```
eventtype TemperatureEvt {
    public float temp ;
};
```

Figure 19: Definition of an Event Type in OMG IDL3 using `eventtype`

Then, the mapping of an event would not be different from the one we have presented here. The mapping from the `eventtype` definition and the consumer interface are then defined as depicted in Figure 20.

```
valuetype TeperatureEvt :
  Components::EventBase {

    public float temp ;
};
interface TemperatureEvtConsumer :
  Components::EventConsumerBase {

  void push (in TemperatureEvt evt) ;
};
```

Figure 20: Mapping of an Event Definition using `eventtype` to OMG IDL2

As when dealing with receptacles, ports related to events lack structural introspection means. It is not possible to discover asynchronous ports offered by a component dynamically without using the Interface Repository and then unfortunately to sustain the complexity of this approach.

3.5 Component Home

The various concepts of the component abstract model have been discussed and it is now possible to define a whole component type. This section presents the component managers that provide instantiation of component types at runtime. The main goal of this section is to describe the reification of the instantiation mechanism, the equivalent of the `new` operator in object oriented models.

3.5.1 Concept

Like `component`, `home` is a new meta-type defined in the context of the CCM. A component home is a manager for instances of a given component type. It manages instance life cycle, using primary keys for indexing purpose. In that, it offers component instance factory and finder operations that use the defined primary key. Home interfaces are defined using single inheritance from the base type `Components::CCMHome`. Classical attributes and operations are also supported in such homes.

A component type is defined independently from home types. But, home type definitions have to specify the managed component type. In the meantime, many home types may manage the same component type, however not the same instances. At runtime, a component instance is managed by a single home instance.

Primary keys are unique identifiers of component instances that take part in management of instance indexing. They are `valuetype`s inheriting from `Components::PrimaryKeyBase`. Such keys have to be a concrete type with at least one public field and no private ones. A primary key type cannot include references of CORBA interfaces nor transitively if the `valuetype` contains structures or unions.

3.5.2 OMG IDL3 Definition

The component home definition (see Figure 21) declares a home type for the `VendingMachine` component type using the `VendingMachineKey` primary key type. In addition to the base operations implicitly defined, the `VendingMachineHome` type includes two operations defined by the designer. These two operations are based upon the *factory* and *finder* design patterns [8]. The definitions are associated with two keywords, respectively `factory` and `finder`. In that, these design patterns associate semantic rules to the operations.

```
valuetype VendingMachineKey :
  Components::PrimaryKeyBase {
  public long identifier ;
};
home VendingMachineHome
  manages VendingMachine
  primaryKey VendingMachineKey {
    factory create_vending_machine
      (in long id)
      raises (Components::DuplicateKeyValue,
              Components::InvalidKey) ;
    finder search_vending_machine
      (in long id)
      raises (Components::UnknownKeyValue,
              Components::InvalidKey) ;
};
```

Figure 21: Component Home Definition Using a Primary Key

3.5.3 OMG IDL2 Mapping

The mapping of home definitions is structured in three parts. The explicit interface groups operations declared by the designer (`create_vending_machine` and `search_vending_machine` that are application operations to be used in order not to use the generated and technical ones). The implicit interface groups generic operations of component homes for this home type. The final interface of the home type inherits from the two previous ones as depicted in Figure 22.

3.5.4 Using Homes

Once home definitions are compiled, various operations are generated for the component type managed by the home. In the context of a keyless home, the only generated operations is `create`. While in the context of a home with a primary key, the operations `create`, `find`, and `destroy` are generated. Finally, operations of the explicit

```
interface VendingMachineHomeExplicit : Components::CCMHome {
  VendingMachine create_vending_machine (in long id)
    raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
  VendingMachine search_vending_machine (in long id)
    raises (Components::UnknownKeyValue,Components::InvalidKey) ;
};
interface VendingMachineHomeImplicit {
  VendingMachine create (in VendingMachineKey key)
    raises (Components::DuplicateKeyValue,Components::InvalidKey) ;
  VendingMachine find_by_primary_key (in VendingMachineKey key)
    raises (Components::UnknownKeyValue,Components::InvalidKey) ;
  void remove (in VendingMachineKey key)
    raises (Components::UnknownKeyValue,Components::InvalidKey) ;
  VendingMachineKey get_primary_key (in VendingMachine compo) ;
};
interface VendingMachineHome :
  VendingMachineHomeExplicit, VendingMachineHomeImplicit {
};
```

Figure 22: OMG IDL2 Mapping of a Home Definition Using Primary Keys

interface have to be implemented by the developer.

Home operations are used to find component instances. To find back component homes, search interfaces (finder) are defined. They allow one to find component home instances using criteria such as a given component type, a given home type, or according to home names in the name service. A home finder is reachable through a call to the `CORBA::ORB::resolve_initial_references` operation using `ComponentHomeFinder` as service name.

3.5.5 `Components::CCMHome` and `Components::KeylessCCMHome` Interfaces

`Components::CCMHome` interface is the base interface of any component home. Any mapping, for keyless home or home with a primary key, inherits from this interface through its explicit one. In addition, keyless homes inherit from `Components::KeylessCCMHome` through their implicit interface. In a dynamic context, the operations `get_component_def()` and `get_home_def()` provide access to component home type and to the component type it could create, as defined in the Interface Repository. These definitions allow home type introspection, in order to know the exact name of the typed operations available to create instances as well as the operation parameters (see Section 3.5.3). In the context of keyless homes, introspection is not required as the `create_component()` operation is always available.

3.5.6 Review

The use of component homes, inspired from EJB, is an important aspect to instantiate components in a generic context: It avoids to write a dedicated program for each component type. Home definitions nicely summarize the philosophy of the CCM: Extract patterns from CORBA 2 programming in order to use them automatically and to generate technical code [29]. In the context of the CCM, homes are characterized and designers may define specific operations associated to a home. Using EJB this is not possible as home interfaces are automatically generated based upon the Bean definition. Finally, we regret that it is not possible to define component homes that are components, as well as it is impossible to define component types that are also home types. This would support hierarchical composition models.

3.6 Component Configuration

3.6.1 Configuration Phase

The life cycle of a component instance is composed of two steps in mutual exclusion: The configuration phase, initialization of the instance, and the

```
module Components {
  interface CCMHome {
    CORBA::IRObject get_component_def () ;
    CORBA::IRObject get_home_def () ;
    void remove_component
      (in CCMObject comp) ;
  };

  interface KeylessCCMHome {
    CCMObject create_component () ;
  };
};
```

Figure 23: `Components::CCMHome` and `Components::KeylessCCMHome` Interfaces

functional one, client use of the instance. It is important to have distinct interfaces for each phase. Nevertheless, this distinction is not clearly stated in the CCM specification. Thus, it is up to the designer to make this separation of concern. He has to explicitly implement the configuration phase of an instance or to provide means to realize it. In the context of the CCM, a component configuration mainly relies on the setting of its attributes.

3.6.2 Configuration Objects

A configuration object encapsulates the configuration of some or all of the component attributes and that is to be used to configure any instance of the component type it is related to. A configuration object may use any operation of the component available at configuration time. Such an object inherits from the base interface `Components::Configurator`, depicted in Figure 24. It provides the `configure()` operation that takes as parameter the component instance to be configured.

The `Components::StandardConfigurator` interface defines a standard configuration object. It provides the `set_configuration()` operation to initialize the value of the attributes to be configured. The attribute values are given as a sequence of couples (attribute name, value) which are defined as a `Components::ConfigValue valuetype`. The `configure()` operation of the configuration object mainly applies values contained in the sequence to the corresponding attributes. This part of the interface is used at deployment time to initialize the configuration objects.

```
module Components {
  interface Configurator {
    void configure (in CCMObject comp)
      raises (WrongComponentType) ;
  };

  valuetype ConfigValue {
    public FeatureName name ;
    public any value ;
  };
  typedef sequence<ConfigValue>
    ConfigValues ;
  interface StandardConfigurator :
    Configurator {

    void set_configuration
      (in ConfigValues desc) ;
  };
};
```

Figure 24: `Components::Configurator` and `Components::StandardConfigurator` OMG IDL Interfaces

3.6.3 Factory Based Configuration

Factory operations contained in component homes may also participate to the configuration process of component instances. In that, a factory operation may:

- be implemented for a specific configuration,
- apply a configuration object to all the component instances it creates, and
- explicitly invoke `configuration_complete()` to finish the configuration phase of an instance.

The implementation of a component home may support the `Components::HomeConfiguration` interface, that inherits from `Components::CCMHome` (see Figure 25). This interface is mainly intended to be used by agents in order to deploy a component instance in a container. This interface lets the user specify a configuration object and a sequence of `Components::ConfigValue` objects. A user of `Components::HomeConfiguration` may also remove the possibility of future use of automatic configuration by the component home.

```
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 () ;
  };
};
```

Figure 25: `Components::HomeConfiguration` OMG IDL Interface

3.6.4 Review

The fact that configuration and use phasis of a component instance are in mutual exclusion is really important. The use case defined here is thus a nice evolution. Nevertheless, the fact that this is only mentioned in the specification is upsetting. Right now, it looks like "it would be good to do so" more than "you have to do it". It is up to the designers to define a configuration interface, which may be a drawback to leave as much freedom on a so sensible aspect. In that, using a component instance badly or not configured at all may result in an unexpected or dangerous behavior (like a system crash).

3.7 Abstract Model Review

The abstract model announces an important evolution of the CORBA object model. It represents the most finalized part of the CCM specification. Its expression means are much more complete than the CORBA 2 object model ones and it allows designers to characterize a component type in its whole: Not only provided services but also required ones and configurable properties.

Dealing with configurable properties, raising exceptions when attributes are misused contribute to build reliable software. But, the lack of mapping from OMG IDL2 to programming language operations in order to use and manage exception looks like an omission. In that, only CORBA 3 ORBs may manage attributes properly, which using constraints would have to be implemented manually in the functional code. The ORB only manages exception marshaling and unmarshaling. This also raises a problem of compatibility: How CORBA 2 ORBs would inter-operate with CORBA 3 ones when using such attributes?

Regarding services provided by a component type, facets offer designers a means to define multiple views on the component type and to best match the needs of the various clients. Introspection capabilities permits users to discover dynamically services provided by a component instance, and then to choose the best point of view according to the expected use.

When a component type client is a component itself, receptacles provide a means to compose those two types together in a satisfying way. They also allow connections between component instances to evolve at runtime. A shadow in this picture is that multiple receptacles semantic is not clearly specified in the abstract model.

The event model is interesting and seems to be of wide use. It is important to notice that the model does not force a particular implementation, many technical choices may be made: List of listeners, event channel, multicast... Thus, the design phasis may be independent of technical constraints and a single event definition may be used in several contexts.

On the drawbacks' side of the abstract model, it is upsetting that there are no structural introspection means, in addition to the Interface Repository, on receptacles and event sources / sinks. A structural introspection API offered by the component would bring simplicity in its use. Another lack about introspection is the possibility to find back the runtime environment of a component instance (container or server). This would allow reconfiguration of the environment at runtime and to optimize instance deployment regarding proximity of existing ones.

A second bad point, which is not limited to the abstract model (as also regarding the execution model discussed later on), is the poor support of the dogma "configure and use are two distinct actions that should exist in mutual exclusion, and furthermore using distinct interfaces." The abstract model claims this point without forcing it by specifying, for example, a base interface provided by component instances for their configuration. Then, the execution model would have to define the use of this interface in a reliable and systematic way.

Finally, the fact that facets are co-located may be seen as a limit. From an application point of view, the abstract model is flat (only one level). Thus, it is not possible to define *composites* (component aggregations) which would be themselves components with distributed facets and thus usable in the same way. Allowing de-localized port would allow the design of composites.

4 Implementing Components

4.1 Programming Model

The programming model offers component providers a mean to produce the implementation of component types. The main goal is to describe non-functional aspects in order to automatically generate this part of the implementation and to only write the functional part of the component. In order for these two parts to be integrated nicely, the CCM provides a framework, the *Component Implementation Framework* (CIF) which describes how functional and non-functional part should interact together.

In order to generate component skeletons, which are extended compared to CORBA 2 ones, the CIF relies upon the *Component Implementation Definition Language* (CIDL). Component skeletons automate the management of component basic behaviors like port management, navigation between facets, and component life cycle. In addition to the CIDL, the CIF relies on the framework defined for the Portable Object Adapter (POA) hiding most of its complexity in the same time.

4.1.1 CIDL and Executors

The *Component Implementation Definition Language* is intended to describe the implementation structure of a component as well as to describe its persistent state. In that, a component is considered as a set of behavioral elements provided by *executors*. Two kinds of executors are defined: Component executors, which implement component types, and home executors, which implement component homes. A component implementation is composed of an aggregation of elements with specific behavior and relationship. CIDL is used to define this aggregation which is named a *composition*. composition is a CIDL meta type corresponding to an implementation definition.

```
composition entity VendingMachineImpl {
  home executor VendingMachineHomeImpl {
    implements VendingMachineHome ;
    manages    VendingMachineImpl ;
  };
};
```

Figure 26: Simple CIDL Definition

A composition, as depicted in Figure 26, specifies a home type (defined in OMG IDL3), and implicitly a component type (as a home type is dedicated to a component type). Then, an executor definition is associated to the home type. This definition specifies the relationship between the home executor and the other elements of the composition. Finally, the composition specifies a component executor definition. This kind of executor may be segmented, *i.e.* its implementation will be split in several classes, implementing facets, with distinct persistence state. The abstract state of a component may also be defined in the context of persistent requirements. The relationship between home type, component type, and executors are described in Figure 27.

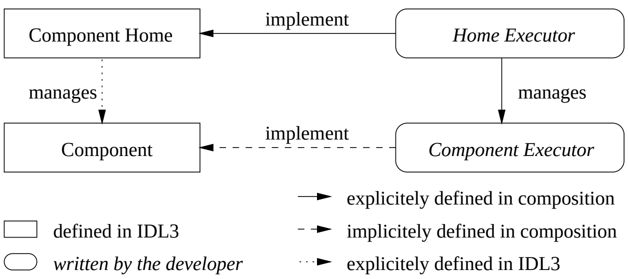

Figure 27: Relationship Between Home, Component Type, and Executors

Four kinds of components are defined. They are depicted in Table 1.

Service components are stateless and without identity. Their life cycle is related to the processing time required by the invocation of an operation. Factory is the instantiation design pattern used by the client. This kind of component implementations may be created on demand and be managed as a pool of component instances.

Component	Container (impl)	Container (kind)	External Identity
Service	Stateless	Transient	Keyless
Session	Conversational	Transient	Keyless
Process	Statefull	Persistent	Keyless
Entity	Statefull	Persistent	With key

Table 1: Component Kind Definitions

Session components state is transient and their identity is not persistent. The life cycle of such a component is the sequence of operation invocations required by the client. This kind of components is suitable to implement iterators or any similar component. Factory is also the instantiation design pattern used.

Process components are statefull and persistence is managed by the component itself or by the container. Clients are unaware of the persistent state of the component. The identity of the component is also persistent and managed by the container. It is visible to the client through operations defined by the designer. Factory is still the instantiation design pattern used. This kind of components is used to define components that represent business processes more than entities.

Entity components are similar to process components dealing with persistence. Nevertheless, clients are aware of the persistence. Such component identity is visible through the use of a primary key specified in the definition of the component home. Design patterns used are both factory and finder ones.

4.1.2 Mapping

The compilation of OMG IDL3 / CIDL definitions describing a component type provides several elements (see Figure 28). First, interfaces for the component clients are defined in OMG IDL2. Second, an XML descriptor is generated which lists the technical constraints to be guaranteed at deployment time. Finally, the skeleton is generated and manages the non-functional part of the component.

Component skeletons implement the OMG IDL2 mapping of a component type definition. It provides structural introspection on the component, discovering ports, as well as managing component connectivity: Connection and disconnection of a receptacle, subscription and unsubscription from an event source, and so on. The skeleton also manages component instances life cycle, its identity, etc.

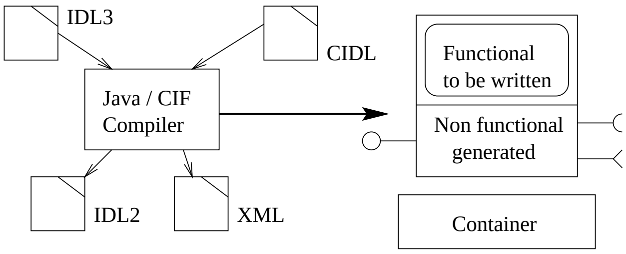

Figure 28: Component Skeleton Generation

4.1.3 Review

At the moment, the CCM specification does not include the definition of the interface between the functional and the non-functional part of a component. In the meantime, there are examples in the specification that give an idea about a set of interfaces to provide / use when writing the executor of a component. This point does not forbid the development of a CCM-based application. Nevertheless, in order to have portable component across containers from various providers and to ease for developers switching of one platform for another, these interfaces should be standardized. This point should be resolved in the next release of the CORBA Component Model specification, based upon the experience of the first CCM implementations. Finally, the fact that the CIDL may include the persistent state of a component is raising issues. The *Persistent Service Definition Language* (PSDL) already exists for such purpose, thus the CIDL should use the PSDL for the persistent aspect. The separation between both languages is not clear at the moment.

4.2 Implementing Components Using OpenCCM

At the moment, the OpenCCM platform do not take into account the CIDL descriptions. The compilation process is only based upon the OMG IDL3 definitions.

4.2.1 OpenCCM Compilation Process

The compilation process of a given language is traditionally made of a lexical and syntactic parser that builds an abstract syntactic tree upon which semantic checking is performed. Then, this tree is parsed to generate code or for direct interpretation / execution.

In our context, the OMG IDL3 is defined using about 250 syntactical rules. The abstract tree has to model more than 20 syntactic constructions, mainly `module`, `interface`, `valuetype`, `component`, `home`, as well as type, operation, attribute, and port definitions. The new `import` statement[4] implies to have access to the abstract tree composed of previous definitions. Semantic checks are numerous and mostly recursive. Finally, generation from the OMG IDL3 includes the mapping to OMG IDL2 as well as extended skeletons for programming languages such as Java, C++, and IDLscript [21].

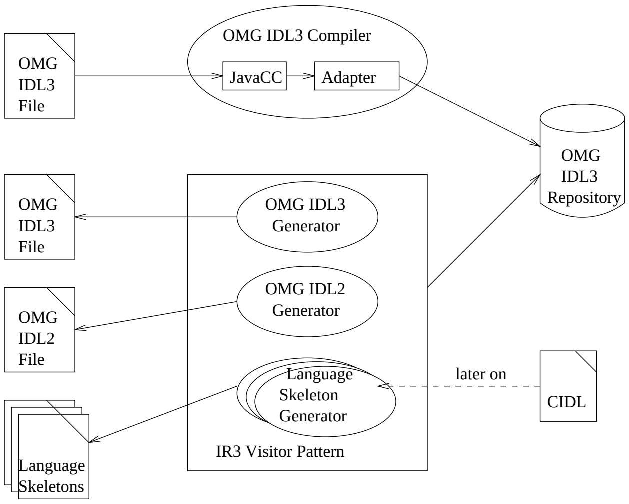

Figure 29: OpenCCM Compilation Process

In order to manage this complexity, the OpenCCM compilation process is made of a set of components depicted in Figure 29. This process is based on the use of an OMG IDL3 repository fed by a compiler of OMG IDL3 files and visited by a set of generators to produce OMG IDL2 and extended skeletons. This architecture allows us to isolate problems which ease the support and evolution of the OpenCCM platform: The compiler only focuses on the lexical and syntactic aspects (implemented with JavaCC [17]), the repository stores abstract trees and performs semantic checks, and each generator only includes the mapping rules of its target language. Generators are build based on a similar approach: The *Visitor* [8] design pattern is used to visit the OMG IDL3 repository. Generators also share a framework to produce output files. This eases the definition of new generators without modifying the whole architecture of the compilation process.

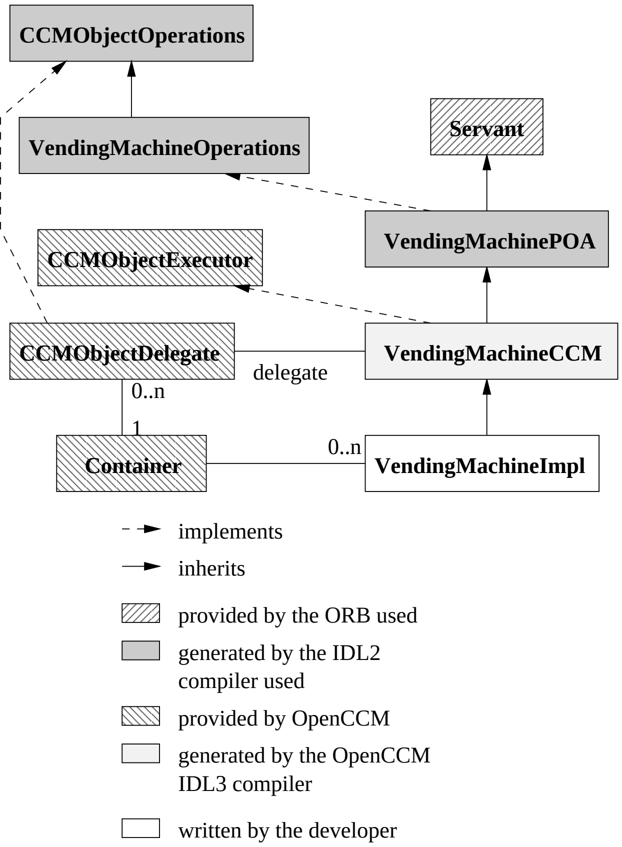

Figure 30: Architecture of OpenCCM Extended Skeletons

[4]OMG IDL3 import nicely replaces the include statement.

4.2.2 OpenCCM Extended Skeletons

Figure 30 depicts the architecture of an extended skeleton generated by the OpenCCM platform. These classes came from three distinct sources.

- `Servant` is provided by the ORB while `VendingMachinePOA`, `CCMObjectOperations`, and `VendingMachineOperations` are generated using its OMG IDL2 compiler.

- `CCMObjectExecutor`, `CCMObjectDelegate`, and `Container` are provided by OpenCCM while `VendingMachineCCM` is generated using its OMG IDL3 compiler.

- `VendingMachineImpl` has to be written by the component developer.

In the context of OpenCCM, the choice of cooperation between the non functional part of the extended skeleton and the functional part is performed through inheritance. `VendingMachineCCM` implements the `VendingMachineOperations` interface and defines callback operations (`_get` operations). `VendingMachineImpl` has to implement the callback operations and may overload the generated implementation. The cooperation between the container and the extended skeleton has not been defined yet as OpenCCM only includes basic containers that do not include processing for the moment.

4.2.3 Excerpts of Code

The class `VendingMachineImpl` represents the executor to be implemented for the vending machine component. In the following, executors are monolithic: All the facet operations are implemented in the same executor class. Callback operations depicted in Figure 31 are OpenCCM dependent and permit the extended skeleton to access the implementation of facets and sinks. This choice is long term as a non-monolithic executor will also be designed using these callbacks. This approach is intended to be standardized in a future release of the specification. It would be nice not to have to write this part of the code. However, without a means to describe the structure of the executor, this part of the implementation is still necessary to be hand written.

The code depicted in Figure 32 is the functional implementation of the `ProviderFacet` of

```
public class VendingMachineImpl
  extends VendingMachineCCM
  implements ClientFacetOperations,
             ProviderFacetOperations,
             RepairmanFacetOperations,
    TemperatureEvtConsumerOperations {

  public ClientFacetOperations
  _get_facet_client ()
  { return this ; }

  public ProviderFacetOperations
  _get_facet_provider ()
  { return this ; }

  public RepairmanFacetOperations
  _get_facet_repairman ()
  { return this ; }

  public
  TemperatureEventConsumerOperations
  _get_consumer_temp ()
  { return this ; }

  /* implementation of the facets */
}
```

Figure 31: OpenCCM Callbacks for the Vending Machine Executor Implementation

the Vending Machine Component. As mentioned above, it is implemented directly in the body of the executor class. The initialization of the variables is not presented here but has to be done. The constructor of the class is a good place to do so as these variables do no rely upon the environment (like other components of system services). This part of the executor is the really important part: The only one that should be written by the component developer, containing the business logic of the component.

Figure 33 presents the implementation of the `configuration_complete` operation that is called to switch the component instance from its configuration phasis to its execution one. This operation also has to be implemented by the developer as it is related to the functional logic of the component. Here, the operation checks if both power and water receptacles have been properly configured. If not, the operation raises an exception to signal that the configuration has not been performed correctly.

Figure 34 depicts the implementation of the heat probe component type. The configuration phase is simple, it just checks if the probe works properly. As this component only produces events and does not use others, there are no connection to check before the validation of the configuration phase. A thread is created as the heat probe is an active component.

Finally, Figure 35 illustrates the implementation of the vending machine home as defined in Figure 21. The static declaration of the class contains the code related to the registration of valuetype factories. In this example, the factories are related to valuetypes used as primary keys and events consumed / produced by the vending machine component type. This choice of a static declarations is OpenCCM dependant. The `create_home` operations is the entry point (see Section 6.1.3) that will be used to instantiate a vending machine home. The operation `create_component_executor` really creates an executor for the component type, its instance at the lnaguage level. The `create_vending_machine` only creates a CORBA reference for the component (without implying the creation of the language instance). This means that one million of component may exists in a server, without consuming

```
public class VendingMachineImpl ... {

  protected boolean   open ;
  protected double    cash ;
  protected double    price ;
  protected Vector    drinks ;

  public void openFrontDoor () {
    this.open = true ;
  }

  public void closeFrontDoor () {
    this.open = false ;
  }

  public double getCash ()
    throws DoorNotOpen {

    if (! this.open) {
      throws new DoorNotOpen () ;
    }
    double tmp = this.cash ;
    this.cash = 0.0 ;
    return tmp ;
  }

  public
  void addDrink (Drink[] s, double p)
    throws DoorNotOpen {

    if (! this.open) {
      throws new DoorNotOpen () ;
    }
    for (int i=0 ; i<s.length ; i++ )
      this.drinks.addElement (s[i]) ;
    this.price = p ;
  }

  /* other facets' operations */

  /* OpenCCM specific methods */
}
```

Figure 32: Provider Facet Implementation of the Vending Machine Component

```
public void configuration_complete ()
  throws InvalidConfiguration {

  if ((get_connection_water() == null) ||
      (get_connection_power() == null)) {
    throw new InvalidConfiguration () ;
  }
}
```

Figure 33: Implementation of the Operation `configuration_complete` for the Vending Machine Component

```
public class HeatProbeImpl
  extends HeatProbeCCM
  implements Runnable {

  Probe probe = new Probe () ;

  public void run () {
    while (true) {
      Thread.sleep (10000) ;
      float d = this.probe.read () ;
      this.push_temp
        (new TemperatureEvtImpl(d)) ;
    }
  }

  public void configuration_complete ()
    throws InvalidConfiguration {

    if (!this.probe.test())
      throw new InvalidConfiguration () ;
    else
      (new Thread(this)).start () ;
  }
}
```

Figure 34: `HeatProbe` Component Implementation (Excerpt)

all the memory. Finally, the two business operations `create_vending_machine()` and `search_vending_machine()` are implemented using the OMG IDL3 operations implemented by the skeleton.

5 Packaging Components

In order to be diffused and deployed, component implementations have to be packaged. Several kinds of deployment packages have been mentioned in Figure 2 page 5. These packages are presented in this section. In the context of CORBA Components, a package is an archive grouping together a component type, one or several implementations of this type, and a set of descriptors. Such descriptors are written in the *Open Software Descriptor* (OSD) language which is a description language defined as an *eXtended Modeling Language* (XML) *Data Type Definition* (DTD) [26]. This language has first been introduced at the W3C by Marimba and Microsoft. Then, it has been extended by the OMG according to the CCM needs. The descriptors are a good basic means to define the architecture of an application. However, they do not have the potential of Architecture Definition Languages (ADL) [1, 4] to which they should not be seen as equivalent.

5.1 Software Package

A software package is made of a descriptor and a set of files, basis of the software entity implementation. These various elements are grouped in an archive using the “ZIP” format. This kind of packages is used to deploy any software package. In the CCM context, a package includes CORBA component implementations.

A descriptor defines the package contents. It provides general information about the piece of software (author, description, OMG IDL interface, and so on) as well as a description of the implementations (name, target OS, implementation language, code localization, etc). This descriptor also enumerates the environment related dependencies of the component implementation. Figure 36 outlines the main elements of the software package descriptor.

```
public class VendingMachineHomeImpl
  extends VendingMachineHomeCCM {

  public VendingMachineHomeImpl () {}

  static {
    DrinkDefaultFactory.register () ;
    VendingMachineKeyDefaultFactory.
      register () ;
    NoChangeEvtDefaultFactory.
      register () ;
    EmptyEvtDefaultFactory.register () ;
    TemperatureEvtDefaultFactory.
      register () ;
  }

  public static
  VendingMachineHomePOA create_home () {
    return new VendingMachineHomeImpl () ;
  }

  public VendingMachineOperations
  create_component_executor () {
    return new VendingMachineImpl () ;
  }

  public VendingMachine
  create_vending_machine (int id) {
    try {
      return super.create
        (new VendingMachineKeyImpl(id)) ;
    } catch (Exception e) {
        return null ;
    }
  }

  public VendingMachine
  search_vending_machine (int id) {
    try {
      return super.find_by_primary_key
        (new VendingMachineKeyImpl(id)) ;
    } catch (Exception e) {
       return null ;
    }
  }
}
```

Figure 35: `VendingMachineHome` Implementation

```
<softpkg>
  <pkgtype>
  <title>
  <author>
  <description>
  <licence>
  <idl>
  <propertyfile>
  <implementation>
</softpkg>
```

Figure 36: Main Elements of a Software Package Descriptor

5.2 Component Package Descriptor

The component package descriptor specifies component characteristics defined in the design and development phases. It is partly generated by the OMG IDL3 compiler and partly modified by a packaging tool (for elements such as persistence, threading, and so on). This descriptor includes as outlined in Figure 37 the various features of the component: Its ports and their interfaces.

```
<corbacomponent>
  <repository>
  <componentkind>
  <eventpolicy>
  <threading>
  <componentfeatures>
    <port>
  <interface>
</corbacomponent>
```

Figure 37: Main Elements of a Component Package Descriptor

5.2.1 Functionality Description

This descriptor is usable through a design tool to present information regarding the component type. It describes the component structure: Inherited interfaces, supported interfaces, ports, and so on. The various interfaces of the component are described and referenced by their Repository Id. This descriptor allows a tool to connect component together at deployment time to build whole or part of an application.

5.2.2 Deployment Information

At deployment time, this descriptor is used to choose the kind of container to be used for the component, and to provide this container the information regarding the component properties like persistence and port quality of service. Information regarding the thread policy to be used as well as the achievement of the configuration phase is also included.

5.3 Component Assemblies

A component package allows the deployment of a component on its own. An assembly package provides a easy means to deploy related components that form an assembly. It is a template / pattern to instantiate a set of components and to connect them together. Such a package contains a descriptor, a set of component packages or their location, and a set of property descriptors. The assembly package is also a "ZIP" archive.

5.3.1 Assembly Descriptor

This descriptor depicts the component assembly, *i.e.* the description of components, connections, and the logical partitioning of instances. Component instances are connected using provides / uses and emits / consumes ports. The descriptor references the files of each component and specify their connections through asserts like `connectinterface` and `connectevent` (which are contained in the `connection` declaration). The logical partitioning of instances will be mapped at deployment time to physical hosts. It represents the basis information of the deployment process. Figure 38 depicts the main elements of an assembly descriptor.

5.3.2 Property Descriptor

It defines the configuration parameters of components and homes. It is used to configure a component instance or a home instance: Setting their attributes. It provides default properties that may be overridden by the user. Such configuration information is used by configuration objects (see section 3.6.2). The property descriptor may contain as many entries as necessary. These entries may be of three kinds: Simple to configure an attribute of a simple type, sequence to configure an

```
<componentassembly>
  <description>
  <componentfiles>
  <partitioning>
  <connection>
    <connectevent>
      <consumesport>
      <publishesport>
    </connectevent>
    <connectinterface>
      <usesport>
      <providesport>
    </connectinterface>
  </connection>
</componentassembly>
```

Figure 38: Main Elements of an Assembly Descriptor

attribute defined as a sequence, and struct to configure composed attributes. Each entry contains the name of the attribute, its type, and its initial value (see Figure 39).

```
<properties>
  <simple name=... type=...>
    <description>
    <value>
  <simple>
  <sequence name=... type=...>
    <simple>
  </sequence>
  <struct name=... type=...>
   <description>
   <simple>
  </struct>
</properties>
```

Figure 39: Main Elements of a Property Descriptor

6 Deploying Components

6.1 Deployment Model

The deployment model is another major contribution of the CCM. It provides means to automate the spreading and the instantiation of a distributed application. It contributes to increase the reuseness of software entities in easing the use and integration of existing components.

6.1.1 Overview

The deployment process is performed through a tool provided by a CCM platform. This tool deploys components and assemblies on target hosts. This deployment application is a client that uses services offered by objects, existing on execution sites, to install, instantiate, and configure components and connections. The five basic steps of deployment are:

1. defining and choosing execution sites,
2. installing implementations where required,
3. instantiating servers and containers,
4. instantiating homes and components, and
5. connecting and configuring components.

These operations are implemented in several objects defined[5] in the CCM specification.

6.1.2 Deployment Process

The deployment process presented in the specification is made of thirteen points (defined at two levels). These points are depicted in Figure 40 and discussed.

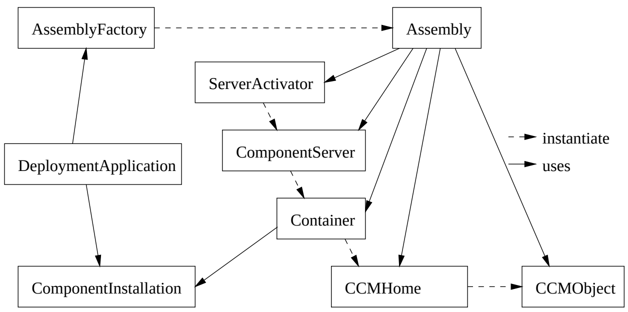

Figure 40: Deployment Process Architecture

1. The deployment application, that uses AssemblyFactory and ComponentInstallation objects, interacts with the user to know the target hosts of the deployment. Once these sites are known, it generates a specialized assembly descriptor from the application descriptor and the information provided by the user. This descriptor is then used by the Assembly instance related to this deployment.
2. Component implementations are installed as archives on required sites. To do so, the deployment tool uses ComponentInstallation instances available on these sites (a single one per site).
3. An assembly factory is used to create an Assembly instance on a single site for the entire application. At creation time, the specialized assembly descriptor URL is provided to the Assembly instance.
4. The assembly descriptor is used as a "recipe" to deploy the application. Based on this descriptor, the Assembly instance creates component homes and component instances on their execution sites.

   To create a component instance, the Assembly object has to create a container server, then a container, then a component home inside the container, and finally the component instance.

   (a) First, the ServerActivator instance available on the execution host is used to create a container server.

   (b) Then, the container server is used to instantiate a container according to the component properties. During this, the container configuration is provided: Container identifier and its properties.

   (c) The container interface provides an operation to install a home. Once installed, the home implementation is loaded using the ComponentInstallation object to know which library to use, *i.e.* the library implementing the component type.

   (d) The Assembly object then uses the home instance to create a component instance. This operation returns the base reference of the component instance as a `Components::CCMObject`.

   (e) If necessary, a configuration object is applied on the newly created instance.

[5] In fact outlined, these objects are under specified.

5. Once all the component instances are created, the Assembly object connects them together using the information provided by the *connection* part of the assembly descriptor[6].

6. If all these steps have been performed properly, the Assembly object invokes on all the instances the `configuration_complete()` operation in order to notify the instances that they are ready to run.

6.1.3 Component Entry Point

Every package contains, at least, one component implementation. The related code is a dynamically loadable module. In order to instantiate a component, a home is required. Component homes are created using an entry point in the module. Its complete name is included in the assembly descriptor in order for the Assembly and container objects to use it. The implementation of entry points is defined regarding the programming language. Figure 41 depicts the two naming rules for the Java and C++ languages which are presented as example in the specification.

```
// Java
public class VendingMachineHomeFactory {
  public static HomeExecutorBase create () {
    return new VendingMachineHomeImpl () ;
  }
}
// C++
extern "C" {
  HomeExecutorBase*
  createVendingMachineHome ()
  {
    return new VendingMachineHomeImpl () ;
  };
};
```

Figure 41: Java and C++ Entry Point for Component Homes

The `HomeExecutorBase` type is the base type of the functional implementation of component homes, *i.e.* the *executor* interface of the home as defined in Section 4.1.1. However, the `HomeExecutorBase` interface is not specified in the specification. Nevertheless, it is discussed at the OMG, based on the first implementations of the CCM. These internal interfaces for the container / component interactions may not be defined in the first release of the CCM specification. But, based on the first implementations, they will be standardized in a second step to guarantee component portability.

[6] This aspect is well defined regarding synchronous ports, but still under-specified or incomplete regarding events.

6.2 Deployment with OpenCCM

The basic steps of the deployment process discussed here are presented using the OMG IDLscript language [21]. They represent the low level operations that will be generated later on from the various descriptors of component archives. Figure 42 depicts the operations of uploading an archive to an execution site (the machine named beck). To do so, it uses the OpenCCM basic container / server to install the archive and to instantiate the home of the component. Then, the script invokes the home operation `create_vending_machine` on the home to create a component instance. This has to be done for each component instance of the application. If an archive is already existing on a site, then the server will not upload it again.

```
# retrieving the name service reference
ns = CORBA.ORB.resolve_initial_references
  ("NameService")
# retrieving the component server
beck = ns.resolve ([["beck",""]])
# installing the component archive
beck.install
  ("vendingmachine",
   "http://www.lifl.fr/vend_machine.jar")
# creating the home instance
home = beck.create
  ("VendingMachineHomeImpl",
   "homeVendingMachine")
# creating the component instance
vm = home.create_vending_machine (123)
```

Figure 42: Creating Component Instances

Once all the instances have been created, they are connected together using the operations provided by the skeleton to retrieve facets and to connect them to receptacles. Figure 43 depicts the connection of the power supply (component instance referenced by `ps`) using the `plug` facet to the vending machine (component instance referenced by `vm`) receptacle `power`. Then, the temperature event sink `temp` of the vending machine is subscribed to the heat probe (component instance referenced by `hp`) source event `temp`. These operations have to be done for all the connections existing in the application. Component instances are also configured through the setting of their attributes.

```
# facet / receptacle connection
ref = ps.provide_plug ()
ck1 = vm.connect_power (ref)

# event subscription
ref = vm.get_consumer_temp ()
ck2 = hp.subscribe_temp (ref)
```

Figure 43: Connecting Component Instances

Finally, when all instances are configured, the `configuration_complete` operation is invoked to start the execution of the application. Figure 44 depicts this final step of the deployment process. It is important to notice that the order of the invocations is important. The vending machine instance should not be running while power and water supplies are not. The order has to be defined regarding component dependencies: Beginning with the instances not depending on others and finishing by instances that are the most dependent of others. This point is unfortunately not outlined in the specification, and right now the only way to defined it is through ordering the definitions of instances of an assembly in the assembly descriptor. But this seems a fairly poor choice.

```
# starting power supply component
ps.configuration_complete ()
# starting water supply component
ws.configuration_complete ()
# starting heat probe component
hp.configuration_complete ()
# starting vending machine component
vm.configuration_complete ()
```

Figure 44: End of the Configuration Phasis

7 Executing Components

7.1 Execution Model

7.1.1 Overview

A container server is an execution environment for CORBA component instances. It provides low level resources such as memory and CPU as well as high level one such as common object services. To provide a generic environment that provides various kinds of contexts to component instances, a server hosts one or several containers that will be know to component instances.

7.1.2 Container Characterization

Component instances, whatever their types, are managed by a container. A component instance cannot not live without being supported by a container. Moreover, a container kind could only host a single component kind for which it has been designed. Two categories of containers have defined: Transient ones and persistent ones.

A container provides a standard set of services to component instances it hosts. Components and homes are deployed in containers through the use of specific tools. Such tool are used to generate useful extensions for the container to host them. A programming API intending to ease application development is specified in the context of the container framework (see Section 4.1).

7.1.3 Container Server Architecture

A container server is a process (in the Unix meaning) that hosts an arbitrary number of containers and components. A manager is responsible for the creation and destruction of containers. It is a container factory accessible at deployment time, according to the component requirements. Each container type is related to an implementation which implies a predefined way of interacting with the POA and the ORB. Each container type includes a specialized POA (regarding for example policies). Container and POA creation, policy configuration, CORBA services binding and so on are defined based on the deployment descriptor.

7.1.4 Portable Object Adapter

A POA is used to create references to be exported to clients. It is also used to manage instance acti-

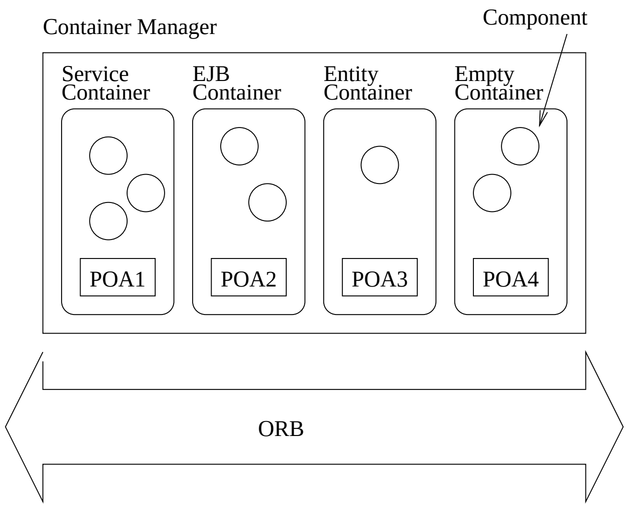

Figure 45: Component Server Architecture Example

vation when requests are received. Creating a container implies most often the creation of a POA that this container will use. This creation is decomposed in four steps:

1. creation of the POA policies defined by the container implementation type,

2. creation of the POA using the policies,

3. assignment of a `ServantManager` to the POA, and

4. POA activation.

For a component installation, a handshake is required between the component and the container. This phase is used to share references between the component and the container, and is performed by the executor base.

7.1.5 Review

The main critic on the execution model is that some points are only partly specified. As the specification is not based on existing implementations, some aspects have never been realized. Thus, most of the interfaces related to containers are only mentioned in the specification but not standardized. Right now, a component will not be portable between two containers of different providers. This aspect is one of the major concern of the first revision of the specification.

Finally, today's vision of containers result in static and non flexible execution support. The number of defined containers is fixed and the set of non-functional properties is static and non extensible. Nevertheless, using adaptive containers would be an important aspect. Then, it would be easy to provide new services or to add services to existing containers. Moreover, containers defined as component assemblies would ease their deployment in container servers according to applications needs. Current work on OpenCCM addresses adaptive containers.

7.2 OpenCCM Basic Container

Right now the OpenCCM platform does not differentiate containers from servers. This choice of a basic monolithic server / container was first intended to quickly provide a runtime support. This architecture will evolve as a generic server for each programming language and real containers will be defined and implemented. The current runtime is a daemon that runs on any potential execution site of the system that will host components. It right now provides two interfaces: An interface to upload archives that is the `ComponentInstallation` one, and an interface that allows the instantiation of homes through the use of their entry points.

8 Conclusion

In the spirit of the OMG, the CCM results of a work leaded by various actors of the worldwide computing area (providers and users) to define a multi-provider heterogeneous distributed component model. Globally, the CCM represents a nice step forward compared to other industrial models such as COM and EJB. However, it is not perfect and work still can be done to improve it, based upon implementation experience.

8.1 Strong Points

Contributions of the specifications are undeniable. First, the CCM provides a global view of the building process of distributed applications. For each step of the life cycle of an application, the CCM brings technical solutions: OMG IDL3 for designers, CIF / CIDL for developers, OSD for architects and deployment, as well as a first shot of the execution environment.

Then, the CCM provides a set of means to describe instead of to program. These description means are targeting provided services definition, like already available in object oriented models, as well as technical and non-functional requirements which is an improvement. Once the various aspects specified, information is generated to be used later on to introspect components.

Finally, in the context of the CCM it is possible to focus on the functional aspects of software components. The system complexity is hidden beyond simplified interfaces provided by both containers and generated part of a component. The later is possible thanks to the previously mentioned description means that allow one to describe non-functional aspects instead of programming them, and to generate the non-functional part of the component.

8.2 Drawbacks

The main drawback of the specification right now is its youth. In that, we are too close for a proper view and there are not much experiment on top of it. Moreover, the goal of the CCM is ambitious and some of the point may be hard to implement. It is also impossible to master all the aspects of the specification from the execution level to the meta-model of the CCM. The process right now is to remove points that are known to be un-implementable. Thus, a real and implemented specification should arise.

From an academic point of view, CORBA components are monolithic, ports are co-located and cannot be distributed over containers for a single component. We also regret the lack of aggregation means: A component assembly is not a component and thus cannot be used as a component. Port definition is static and no port may be added on a component type dynamically. This last remarks is related to the lack of aggregation means, it argues about the lack of dynamic aggregation.

Then, the Component Implementation Framework is a purely technical framework. The CCM does not provide a component design framework. This lack raises some questions: Is it easy to implement components? When should I use a component, when should I keep using objects? Does every thing should become a component? *Could* every thing become a component?

Finally, it seems that the CCM properly defines how components for distributed applications should be produced. Nevertheless, it does not provide guidelines on how to produce, using these components, distributed applications. No rule nor methodology is outlined.

Whenever, these drawbacks, the CCM looks like the most complete component model and will certainly be one of the major industrial model of the forthcoming years. It will not be the ready to use solution of the two forthcoming years. But, it is expectable that the model and its associated platforms will be ready in few years when the users discover the limits of models like COM/DCOM, .NET, and EJB which could be qualified of *first generation of industrial component models.*

8.3 EJB *vs* CCM

Table 2 compares, on given aspects, the CCM and the EJB models. According to this table, the CCM can easily be seen as more flexible and powerful than EJB. First, EJB may be seen as a subset of the CCM as a CCM dedicated container may host EJB. Second, EJB is a proprietary solution only usable with the Java programming language. The CCM tends to be open, as any one may contribute, and usable in an heterogeneous environment. Third, EJB are not really distributed as an application could only be deployed to a single site. On the opposite a CCM application may be spread over several sites. Finally, EJB is a technology of today, many products are already available and used. The CCM is a technology of tomorrow, no complete platforms are existing but it should become a major model in a few years according to its potential.

8.4 OpenCCM

OpenCCM was the first early implementation publicly released. It is right now available in Java, and will also be available in the future in C++. Even if it does not provide all the functionalities of the containers defined in the specification, it already provides a means to define, develop, deploy, and run applications based on CORBA components. As an open source project, people are encouraged to participate and to contribute to the development of the OpenCCM platform.

On the perspective side, on-going research projects include the definition of adaptive contain-

Aspect	EJB	CCM
Availability	Many products	OpenCCM, MICO
Connections	Java Bean to implement	Four kinds of ports
Programming	Java	Any language
Packaging	JAR + XML	ZIP + OSD
Deployment	Single server	Multiple servers
Runtime	Container generation	Container configuration
Use	Today	Tomorrow

Table 2: Comparison of EJB and CCM

ers. These containers will be seen as component assemblies. This approach will lead in the support of the four container types of the specification, but also much more. As containers will be adaptive, they will permit users to define their own non-functional properties and to support them in the containers using a framework. Another aspect of our research project is the study of the definition of application architectures in the context of the CCM. This work will include the support of a flexible deployment process for the CCM. It will be able to match application specific requirements that could not be done with the actual specification.

Finally, all the development work realized in the context of the OpenCCM platform represents a valuable knowledge that contributes to the finalization of the OMG specification and to its future evolution. According to this way of working, OpenCCM should always be up to date with the last version of the CCM specification. The OpenCCM archive includes demos to learn how to use the platform. The complete implementation of the vending machine example should soon be added also. The last release of the platform is available at:

`http://corbaweb.lifl.fr/OpenCCM/`

About the Authors

Both authors are member of the GOAL Team of the LIFL (Computer Science Research Lab of the University of Lille, France). They are also members of the OMG and especially of the CCM Finalization Task Force that aims at finalizing the current CORBA Component Specification. As part of their research, they work on the OpenCCM platform, which is the first implementation of the CCM.

Raphaël Marvie has been working on his PhD at the LIFL since September 1999. The subject deals with distributed component based application architecture management, from its definition to its use in deployment and management of applications at runtime. His work aims at defining a meta model of the CODeX environment (*Composite Oriented Deployment and eXecution*) as well as tools to use it in various component model to support dynamic application definitions.

Philippe Merle is assistant professor at the LIFL since 1997. He is the manager of the OpenCCM project. His main interests are around the building process of distributed component based applications, especially component trading as well as their use through scripting. The OMG IDLscript specification (CORBA scripting language) is the direct result of his PhD work around the CorbaScript language, a dedicated CORBA scripting language.

References

[1] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proceedings IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[2] L. Bass and R. Kazman. Architecture-Based Development. Technical Report CMU/SEI-99-TR-007 ESC-TR-99-007, Software Engi-

neering Institute – Carnegie Mellon University, April 1999.

[3] L. Bellisard and M. Riveill. From Distributed Objects to Distributed Components: The Olan Approach. In *Workshop Putting Distributed Objects to Work, ECOOP'96*, Austria, July 1996.

[4] P. Clements. A Survey of Architecture Description Languages. Height International Workshop on Software Specification and Design, March 1996. Germany.

[5] P. Clements and L. Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003 ESC-TR-96-003, Software Engineering Institute – Carnegie Mellon University, February 1996.

[6] A. DeSoto. Using the Bean Development Kit - a Tutorial. *http://java.sun.com/beans/docs/Tutorial-Nov97.pdf*, November 1997.

[7] F. Bachman *et al.* Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, May 2000.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Westley Professional Computing, USA, 1995.

[9] R. Grimes. *Professional DCOM Programming.* Wrox Press ltd., Birmingham, Canada, 1997.

[10] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Westley, 1999. *ISBN: 0-201-37927-9.*

[11] Inprise. VisiBroker 4.5 for Java User Guide, 2001. *http://www.inprise.com.*

[12] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, pages 305–314, Antwerpen, Belgium, September 2000. IEEE. *ISBN: 0-7695-0819-7.*

[13] R. Marvie, P. Merle, J.-M. Geib, and C. Gransart. CCM + IDLscript = Applications Distribuées. *Evolution des plates-formes orientées objets répartis, numéro spécial de Calculateurs Parallèles et Réseau*, 12(1):75–104, 2000. Ed. Hermès, *ISBN: 2-7462-0169-0.*

[14] R. Marvie, P. Merle, J.-M. Geib, and M. Vadet. OpenCCM: une plate-forme ouverte pour composants CORBA. In *Actes de la 2ème Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, pages 1–14, Paris, France, April 2001.

[15] V. Matena and M. Hapner. *Enterprise Java Beans Specification v1.1 - Final Release.* Sun Microsystems, May 1999.

[16] T. Meijler and O. Nierstrasz. Beyond Objects: Components. In *WCOP'98 Proceedings of the Third International Workshop on Component-Oriented Programming*, 1998.

[17] Metamata. Java Compiler Compiler. *http://www.metamata.com/javacc/index.html.*

[18] Sun Microsystems. *Java Remote Method Invocation Specification.* Sun Microsystems, October 1998.

[19] OMG. *CORBAServices: Common Object Services Specification.* Object Management Group, November 1997.

[20] OMG. *CORBA Components: Joint Revised Submission.* Object Management Group, August 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.

[21] OMG. *CORBA Scripting Language Specification, v1.0.* Object Management Group, June 2001. OMG TC Document formal/01-06-05.

[22] OMG. *CORBA/IIOP 2.4.2 Specification.* Object Management Group, February 2001. OMG TC Document formal/01-02-01.

[23] OOC. ORBacus 4.0.5 for C++ and Java User Guide, 2001. *http://www.ooc.com.*

[24] OpenORB. OpenORB 1.0.1 User Guide, 2001. *http://www.openorb.org.*

[25] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Westley, 1999. *ISBN: 0-201-17888-5.*

[26] W3C. *eXtensible Markup Language (XML) 1.0.* World Wide Web Consortium, 1998. `http://www.xml.com/axml/testaxml.htm`.

[27] N. Wang, D. Schmidt, M. Kircher, and K. Parameswaran. Applying Reflective Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, Tapei, Taiwan, October 2000.

[28] N. Wang, D. Schmidt, and D. Levine. Optimizing the CCM for High-Performance and Real-Time Applications. Middleware'2000, Work-in-Progress Session, 2000. New-York, USA.

[29] N. Wang, D. Schmidt, and C. O'Ryan. *Component-Based Software Engineering: Putting the Pieces Together*, chapter An Overview of the CORBA Component Model. Addison-Westley, 2001.

[30] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9, November/December 1996.