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

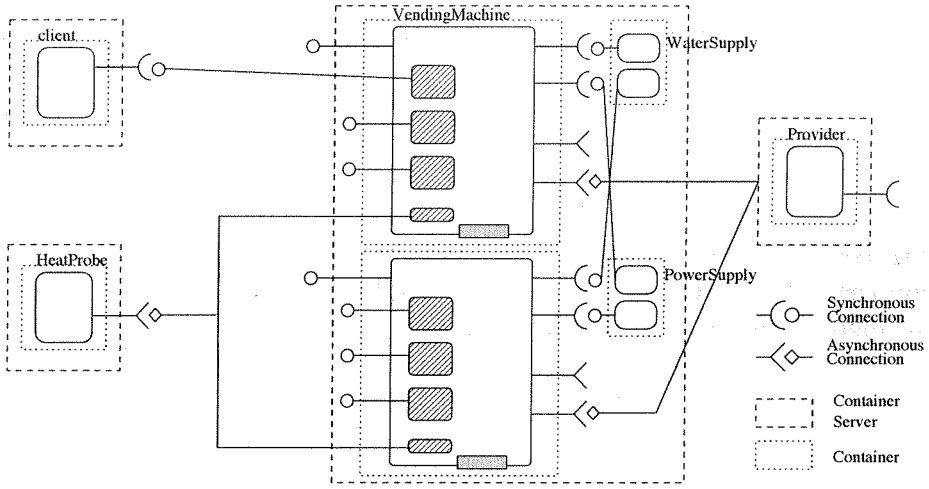


Figure 1: CCM Based Distributed Application

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.

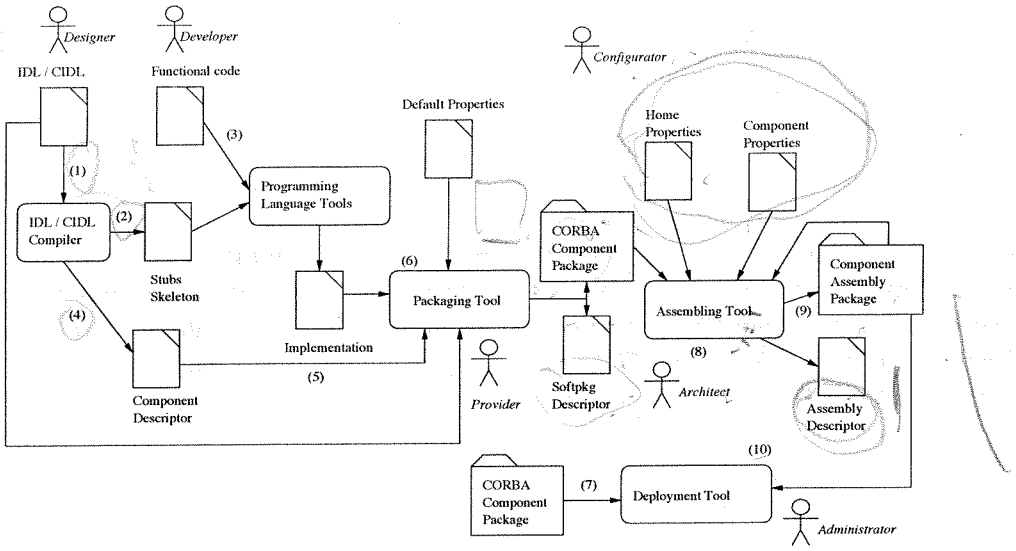


Figure 2: Global CCM Software Production Process

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).

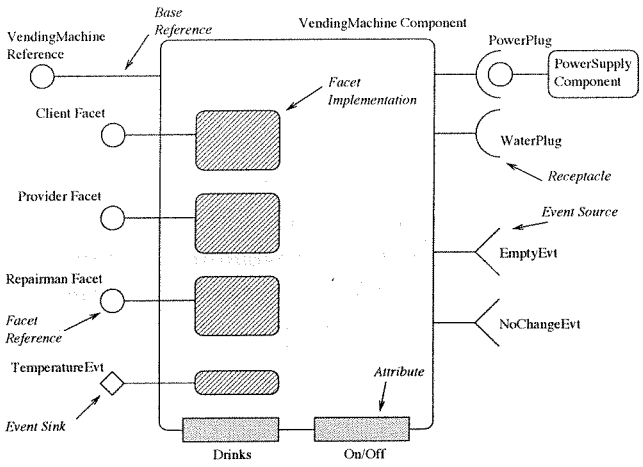


Figure 4: The CCM Abstract Model, Applied to the Vending Machine Example

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.

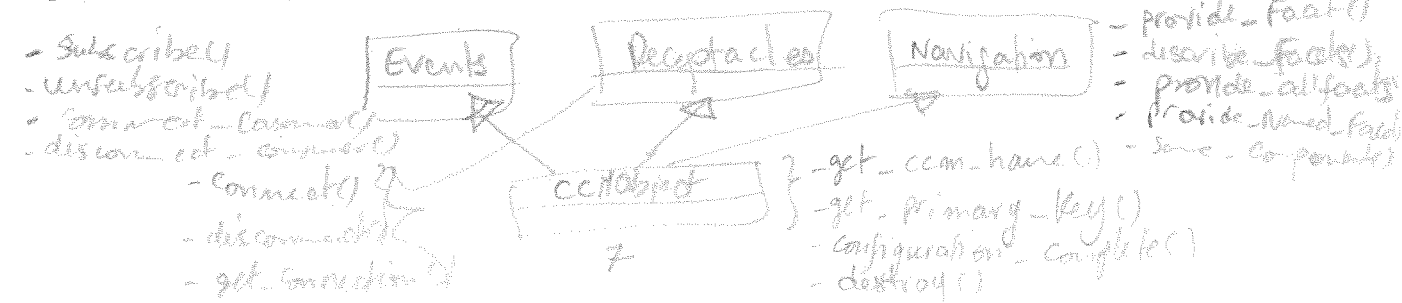
```
interface VendingMachine :
  Components::CCMObject {

  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),



```
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.

```
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.

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

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 valuetypes 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

```
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.

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.

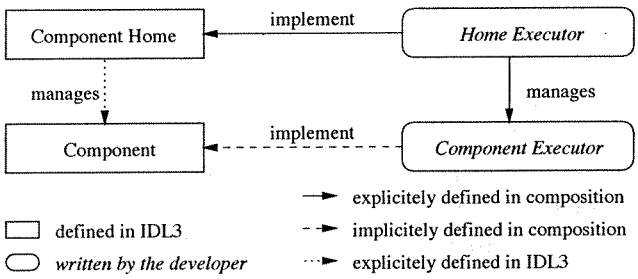


Figure 27: Relationship Between Home, Component Type, and Executors

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.

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].

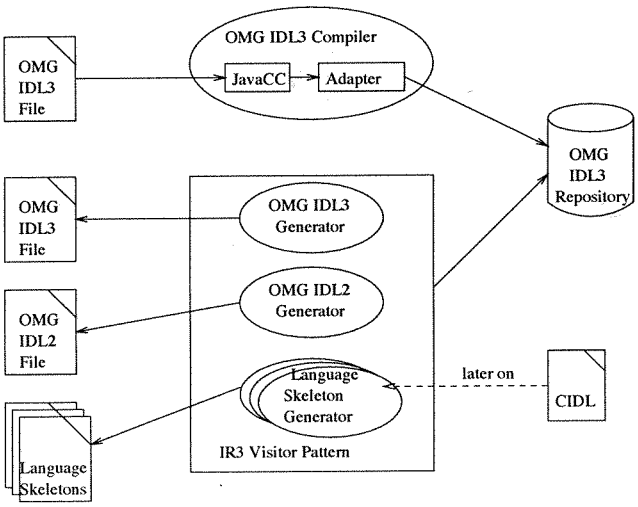


Figure 29: OpenCCM Compilation Process

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.

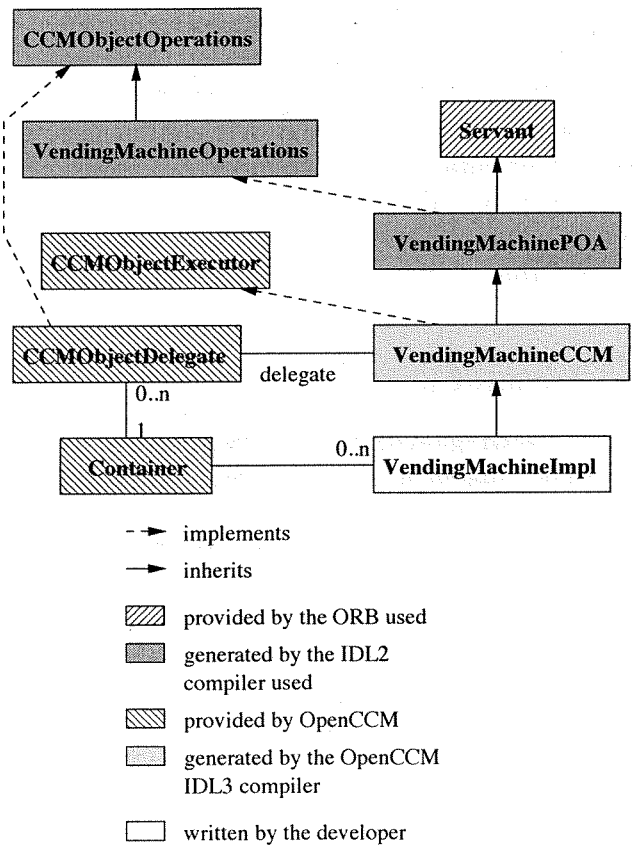


Figure 30: Architecture of OpenCCM Extended Skeletons

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

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 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.

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.

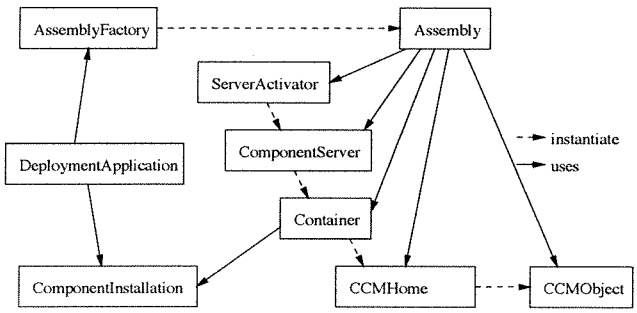


Figure 40: Deployment Process Architecture

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.

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-

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-

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.*