

Architecturing and Configuring Distributed Application with Olan

Roland Balter, Luc Bellissard, Fabienne Boyer, Michel Riveill, and Jean-Yves Vion-Dury

SIRAC project,

INRIA Rhône Alpes

655, Avenue de l'Europe F-38330 Montbonnot, France

e-mail: Luc.Bellissard@inrialpes.fr

Tel: +33 4 76 61 52 78 Fax: +33 4 76 61 52 52

FULL TECHNICAL PAPER SUBMISSION

ABSTRACT

Middleware platforms are today solutions to the problem of designing and implementing distributed applications. They provide facilities for heterogeneous software components to communicate remotely, according to various interaction model, for example client server communication or asynchronous message passing. However, middleware platforms, like message busses or object request brokers, hardly provide tools for the design, configuration and installation of complex software architectures.

This paper presents the Olan environment which primary objective is the description of complex architecture, their configuration according to application requirements and the deployment on heterogeneous distributed environments. The Olan environment is based on an Architecture Description Language where the application is considered as a hierarchy of interconnected software components. A set of software engineering tools assist the application designer in his task of wrapping existing software modules into components which are in turned interconnected through various communication models and finally assembled together according to specific configuration criteria. At run time, a set of system services in charge of automating the installation of the components and their communication channels on a given middleware platform, thus facilitating the overall deployment process of the global application.

1. INTRODUCTION

Middleware technologies are adopted increasingly in software development projects whenever some kind of mechanisms are required to enable remote communication between heterogeneous software components. Among the numerous available middleware, Object Request Brokers such as OMG's CORBA[8] or Microsoft's DCOM[11], are the leading references in the context of industrial development, but other mechanisms such as software busses[10] or message queuing systems are also worth to be considered.

Basically, an ORB-like system provides features for the remote access to heterogeneous software. Integration of

existing applications is realized with the separation of the core implementation and the interface. Interfaces are described in an homogeneous way with an IDL (Interface Description Language) independently of the programming language or the runtime system implementation. At runtime, each middleware platform provide a specific communication pattern; client server synchronous interactions in case of ORBs or anonymous communications based on asynchronous message passing in case of message queues or message busses.

In both cases, such middleware approach suffer from the lack of tools for describing the overall application architecture and to configure the set of software components according to the operational environment. In addition, interoperability between components supported by different middleware is difficult to achieve as it requires for the programmer the implementation of specific gateways.

The goal of the Olan environment is to provide a coherent basis for the construction, configuration and deployment of distributed applications. This environment relies on an Architecture Definition Language (ADL)[1][5][6][12] which describes an application as a set of software components connected through communication objects called connectors. A key idea behind connectors is to separate the description of the components from that of their communication channels with other components. This should allow more flexibility as communications between components can be (re)configured independently from the components, according to functional dependencies or underlying middleware constraints. Compare to other ADL[9][6][12], the Olan environment tries to provide an architectural view as close as possible to the executable image, in particular concerning the use of the communication services.

The Olan environment addresses four issues:

- The integration of legacy software within an application by wrapping them into components,
- The encapsulation of the middleware communication mechanisms in specific objects called connectors,

- The architecture definition, in terms of a hierarchy of interconnected components,
- The deployment of the application (i.e. components and connectors) on a distributed system. Deployment refers to the placement of application components, their installation on nodes and the set up of communication channels between components according to the interconnections properties contained in the architecture description.

Section 2 outlines the main features of the Olan Configuration Language (OCL), the architecture definition language of the Olan environment. Section 3 describes the main functions of the Olan Configuration Machine (OCM), the system service that enables the deployment and installation of applications. Finally, section 4 presents the implementation of the OCM.

2. THE OLAN CONFIGURATION LANGUAGE

The Olan Configuration Language is an Architecture Definition Language specially designed for distributed applications. The targeted applications are in particular those involving multiple users in a distributed environment that cooperate together while using their favorite software, not necessarily designed to work remotely[3]. Those applications require from the configuration language the ability to:

- integrate in an homogeneous way heterogeneous software components,
- have them distributed in various way across the networked computers according to users connections, and
- adapt the communication between components according to the designers' preferences, the components distribution and the middleware characteristics.

In addition, the involvement of multiple users results in unpredictable behavior due to user actions during execution. This requires from the language some facility to express such dynamic behavior.

The design of OCL was initially inspired from Darwin[6] and its notion of hierarchically interconnected components. However, the dynamic behavior specification included in Darwin, as well as the limitation to specify communication properties, have lead us to incorporate new abstractions that will be detailed in this section. OCL defines the abstractions of connectors, representing the objects that rule the communication between components, and that of collections, representing a set of components which can be dynamically controlled in the configuration language. The following sections will present the various features of OCL through an outlook of the required properties inherent to the targeted applications: the facilities to integrate legacy software, to define evolving architectures, and to distribute software components in a flexible way.

2.1 Integration of software

Primitive components are the unit of software integration. They are defined with an interface that describes the

requirements and the provisions of the component. Thus the functional dependencies of the component with the outside world are shown at the interface level. The implementation part specifies what kind of software is encapsulated and where to get the information to have access to it. Primitive components are the basic bricks of an application because they integrate the legacy software modules. The other abstractions of OCL are mainly purposed for architectural issues.

2.1.1 Component Interface

The interface part describes the services that a component provides to other components, along with the services it requires for its execution. Services are described in terms of their signatures in the Olan Interface Language (OIL) which is a syntactic extension of IDL. As interfaces are expected to reflect as much as possible the behavior of the encapsulated software, OIL provides a specific type system which allows to express both the data flow and control flow constraints associated to services.

A specific OIL data type mapping is defined for several programming languages (actually C, C++, Java and Python). This mapping allows translating the data into a pivot format used to perform dynamic conversions between connected services which types can potentially mismatch.

From the point of view of the control flows, the types of provided services defined by OIL are divided in two parts: the classical services, intended to be called synchronously, and the event-based services which can be executed asynchronously in reply to a notification. Respectively, OIL defines two kinds of request that can be made by a component: synchronous service requests, and event submissions. This is summarized in the Table 1.

Services	Data flow	Control flow
Provide ●	in/out parameters	No specific constraints (usually executed synchronously, like procedure calls)
React ■	in parameters	The encapsulated implementation contains an execution flow that handles the service processing in an asynchronous way.
Require ○	in/out parameters	The encapsulated implementation is designed for synchronous external calls (e.g. procedure call)
Notify □	in parameters	The encapsulated implementation is designed for asynchronous external calls (e.g. event notification)

Table 1. Interface Service Types

Exhibiting this information at the interface level is very useful since it characterizes the behavior of components outputs and entry points. In addition, such distinction has led us to define an operational semantics of the language as well

as tools to check the validity of component compositions[13].

```
typedef sequence <*, char> seq;

Interface listBoxItf {
    provide init();
    // services to add, remove or reverse the
    // video of an entry
    provide addEntry(in seq entryContent);
    provide remEntry(in seq entryContent);
    provide revVideoEntry(in seq entryContent);
    // notifications raised when the user
    // manipulates an entry
    notify userAddEntry(in seq entryContent);
    notify userRemEntry(in seq entryContent);
    notify userSelectEntry(in seq entryContent);
    ...
};
```

A simplified example of a component interface, inspired from one of our experiment, is presented above. The interface reflects the notifications and the services provided by a Python module which manages a listBox widget. This module has been integrated in a distributed application from which are taken the different examples given in this paper.

2.1.2 Component Implementation

The component's implementation directly maps the interface specification to the integrated pieces of code. Primitive implementations are made of modules or classes, each of them referencing an integrated pieces of software, either a program source, a library, or an executable. Each module or class is typed according to the kind of software. Modules directly make references to a list of functions whereas classes enable the instantiation of an object when the primitive component is part of the application. For example, modules can be of type «C» or «Python»: a «C» module integrates either C source files or a library, a «Python» module integrates Python source code. If the available classes are written in «C++» or «Java», the primitive components are built from their respective source files. A primitive implementation may contain several modules or classes of the same type. The definition of modules or classes is also described from an interface and it contains information about the location of the files representing the program, the library or the executable, as well as options for their inclusion in the application.

```
module "Python" listBoxMod : listBoxItf {
    path: "${OLAN-SRC}" + "/examples/URL";
    sourceFile : "listboxMod.py";
};
```

The integration of software is greatly helped by the OCL compiler as we will see later. The idea is to capture all incoming and outgoing function calls and use the previously mentioned pivot format. Then, calls from the modules gets out to a component object included in the runtime structures of the application. On the opposite, calls to the software modules gets first to the component object, that knows how to adapt the desired function call to the encapsulated module.

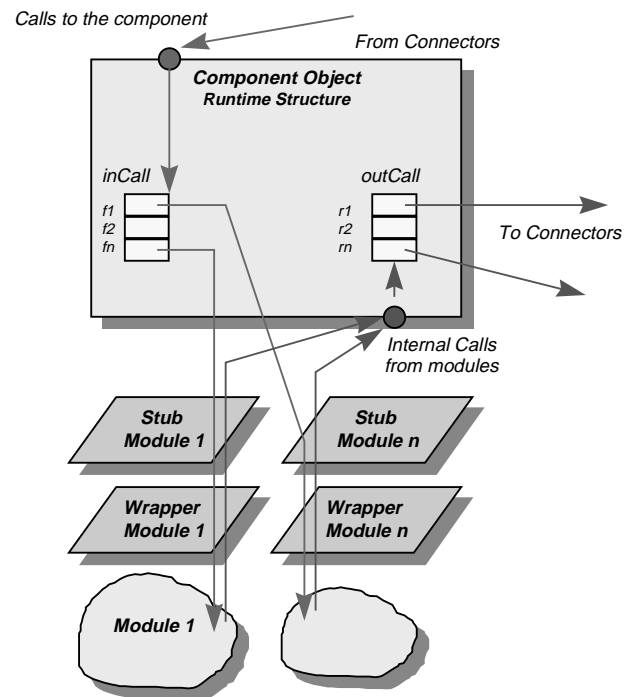


Figure 1. Software Integration at Runtime

Figure 1 shows how things are working at runtime. In between the component object and the various modules, is the *stub*, that homogenize parameters format, and the *wrapper* that knows how to have access to the modules (if the call comes from the component) or the component object (if the call comes from the module). Stub and wrappers are automatically generated by the compiler. According to the kind of modules (or classes), the work to be done by the programmer to have his code integrated, ranges from almost nothing to the explicit redirection of the outgoing calls to the wrapper. The runtime architecture of components and modules will be detailed later in section 3.1.

2.2 Architecture Definition

The definition of an application architecture is basically made of the specification of the instantiation of components and the specification of their interconnection, i.e. the dependencies between required and provided services of their interfaces. The specification of components interconnection can be realized within the implementation of a particular kind of components, the composite.

2.2.1 Application Structure

Composite components are intended to describe the application's structure. They own an interface and an implementation like the primitive ones. The differences between both kind of components lies in the implementation part. A composite implementation contains the specification of a group of components of both kind, that realize the interface of the composite. In addition, the communication requirements between those sub-components is also specified at this level. The interesting facts about composite components is that the application designer can group

components that may have some common properties, like the machine which is intended to host them.

```

Implementation URLNotifier
    uses ListBoxItf, URLMgrItf, URLEditorItf
{
    URLMgr = instance URLMgrItf; // mgt of the URLs
        of interest
    ListBox = instance ListBoxItf; // mgt of the
        user itf
    URLEditor = instance URLEditorItf; // edition of
        URLs

    // expression of interactions between components
    (see 2.2.2)
    startURLNotifier() => URLMgrItf.init()
    URLMgr.initGUI() => ListBoxItf.init()
    ListBox.userAddEntry(URL) => URLMgr.addURL(URL)
    URLMgr.editURL(URL) => URLEditor.edit(URL)
    ...
}

```

As an example, the composite component described here (named URLNotifier) was used in a distributed application that we developed on top of Olan. This application warns users when some of the URL they have previously registered, points to a document that has changed since its last consultation. The URLNotifier component is composed of three sub-components, managing respectively the graphical user interface, the list of URLs of current interest, and the edition of a document associated with an URL (this last component encapsulates a Netscape browser). Let us notice that both the declaration of components instances as well as their interconnections or the interconnections between the composite interface and the components' instance are part of the implementation of the composite component.

The application itself is also a composite component which main feature is to be the root of the component hierarchy.

2.2.2 Interconnections

Interconnections specify how components communicate with each other. By the term «how», we refer to the functional dependencies between interface services, but also to the specification of the data flows, the execution flows, the communication protocol and the runtime mechanism intended to perform the communication. One particular object is dedicated to interactions : the connector.

Connectors are the abstractions in the architecture that specify the interconnections between a set of components. A set of predefined connectors are offered to the application architect (c.f. Table 2). Connectors are named in a unique way, each of which corresponding to a communication pattern and an implementation on top of a middleware platform.

From Table 2, one can notice that synchronous and asynchronous connectors both use ILU; an ORB, for remote communication between objects. However, creating another connector, with the same communication pattern but a different implementation (e.g. sunRpc or TCP/IP sockets), is possible. From the application architecture point of view, there is no difference, as long as the new connector provides a remote synchronous call.

OCL Name	Caller	Callee	Protocol	Implementation
syncCall	1 require	1 provide	Synchronous communication	Local: method call
			Compatible signature or data flow transformation	IPC: ILU method call Remote: ILU method call
asyncCall	1 notify	1 react	Asynchronous Communication	Local: method call from separate threads
			Compatible signature or data flow transformation	IPC : ILU asynch. method call Remote : ILU asynch. method call
RandSync Call	1 require	1 to n provide	Like syncCall, single randomly chosen callee	like syncCall

Table 2. Some predefined Connectors

One important property of connectors, in addition to the above roles, is to manage component heterogeneity and to adapt the control and data flows traversing a set of components. For example, control flow adaptation may be required when one component, requiring a service asynchronously, is bound to another component, providing this service through a synchronous call. The OCL compiler handles such properties and specializes the connector for execution. Data flow adaptation is also an integration problem that often arises when the designer interconnects services with different signatures, notably containing different parameter types. In some cases, the adaptation can be derived by the compiler automatically (e.g. from type string to sequence of char) but it is good practice to specify it within the OCL description.

In the following example, the compiler authorizes the binding if the programmer specializes the connector by specifying the data transformations using a set of operators (boolean operations, concatenations, slicing,...) over formal parameters. In order to maximize the flexibility of the interconnection, the OCL compiler performs static type verifications and generates code for dynamic verification when this is required.

```

typedef sequence<100, char> seq;

interface ListBoxItf {
    notify userAddEntry( in seq URLDesc);
        // bounded sequence of characters
    ...
}
interface URLMgrItf{
    provide addURL( in string URL,in int
        pollingPeriod);
    ...
}

```

```

implementation URLNotifierImpl ... {
...
  ListBox.addEntry(URLDesc) =>
    URLMgr.addURL(URL, pollingPeriod)
    using syncCall() // connector type
    do {
      URLDesc = URLDesc.split(' ');
      URL = URLDesc[0];
      pollingPeriod = URLDesc[1];
    };
...
}

```

2.2.3 Dynamicity

Components and connectors, as described previously, do not allow to express the dynamic behavior which is typical for distributed applications involving multiple users. More precisely, it should be possible to:

- instantiate a given component when the application is running, for example, when a specific function is required by a user,
- express interactions between a set of components whose composition (in terms of components which have been instantiated inside this set) evolves during the execution of the application. For example, within a distributed software development environment, one may want to express that a given message should be received by all development tools that are currently under execution. A similar situation occurs when a distributed application includes a component being in charge of managing user rights to access a given resource: this component should be able to send a notification to all current users.

OCL complies with these requirements through two features: the dynamic instantiation and the collections. The dynamic instantiation allows a sub-component to be instantiated at different times: (1) in the same time as the enclosing component, (2) when the first communication request reaches the component, or (3) when an explicit instantiation order arrives.

The notion of collection is intended to comply with the third requirement listed above. It provides a way to express that a component may include a set of component instances which cardinality may evolve during the execution of the application. A collection is identified by a name and by the type of the components it contains. The evolution of the set of component instances is controlled by a specific connector, bound to the collection. Access to the members of a collection can be performed with an associative naming mechanism which allows to perform a given communication only with the members that satisfy some properties (e.g. their location, attribute values, etc.). Such naming schemes is based on a very dynamic selection of the participants in the communication, like in [7].

```

Implementation URLMgr ... {
  URLPollerColl = collection URLPoller;

  // create a new URLPoller instance for each new
  // URL of interest
  addURL(URL, pollingPeriod) =>
    URLPollerColl.addURL(URL, pollingPeriod)
    using createInCollection();

  //change polling period of a particular
  // URLPoller within the collection.
  // Associative naming is used
  changePollingPeriod(URL, period) =>
    URLPollerColl.changePollingPeriod(period)
    where URLPollerColl.URL == URL
    using aSyncCall();

  // notify all URLPoller instances when the
  // application terminates
  . close() => URLPollerColl.close()
    using broadcastSyncCall()
}

```

With the example of the URLNotifier application, the concept of collection was used to allow the instantiation of an URLPoller component per URL of interest (let's recall that the list of URL of interest evolves dynamically through the user orders). The collection allows the expression of the previously detailed interactions. The first one triggers the creation of a new component within the collection. The second one enables the access to one particular instance of a URLPoller in the collection, whereas the last one sends a message to every component of the collection. The associative naming and various types of connectors are thus used.

2.3 Distribution

In most architecture definition languages, the distribution of components is restricted to the placement on a named computer node[6][9][12]. However, when dealing with the application architecture, an important issue is the distribution of components in terms of processes. Darwin overpasses this issue by considering that every primitive component is associated to one activity (a thread). UniCon goes one step forward, allowing to explicit the association between a component and a process. However, only components affected to processes may be distributed, resulting in a per process basis for the distribution

The ideas developed in OCL are slightly different. The targeted applications mainly imply the cooperation of several users. Placing components in a distributed environment not only deals with the node of execution but also with the user for which the component can execute. In addition, since industrial distributed environments evolve, we have tried to allow the application designers or administrators specifying the placement policies themselves, avoiding the explicit naming of nodes.

The underlying abstraction for placing components is the *context*. A context corresponds to one (or several) execution space that belongs to a single user, on a single machine for a single application. It is uniquely identified by those three items. Then, the distribution specification indicates in which context the component should be executed. At runtime, the

notion of context may correspond to several processes or threads, depending on the kind of software that has been integrated in components. For example, a Python component should be executed in a Python interpreter. The same also applies to a Java component even though both components are specified to be on the same context.

2.3.1 Management attributes

At the OCL level, the distribution and placement policies are described in a special section attached to a component. A policy is a predicate involving the *management attributes* of the component. A management attribute, associated to some component, corresponds to a physical resource that may be used at runtime. The predicate expression is evaluated when the installation of components takes place in order to find a hosting node.

Management attributes are made of several fields, each of which enables to characterize the resource. So far, the existing attributes are Node and Owner, which respectively enable to choose the execution host and the owner of the component execution.

```
management attribute Node {
    string name ;           // DNS name
    string IPAdr ;         // IP address
    string platform ;       // architecture
    string os ;             // OS type (posix, nt, ..)
    short osVersion ;       // OS version number
    long CPUload ;          // average load for the
                           // past 10 minutes
    long UserLoad ;        // nb of connected users
}
management attribute Owner {
    // characterization of the component owner
    string name ;           // user name
    long uid ;              // user id
    sequence<long> grpId ;  // list of groups
}
```

2.3.2 Distribution policies

Policies to express the placement of components are combination of predicates on the management attributes. Such predicates can use various operators (boolean operations, operations on strings, integers, etc.) in order to denote a matching expression for the management attribute which has to be evaluated to true when installing the application.

```
Management URLNotifierMgt: URLNotifierImpl
{
    // All components on the domain .inrialpes.fr
    Host.name == "*.inrialpes.fr";
    // Co-locate the URLMgr and ListBox components
    URLMgr.Host.name == ListBox.Host.name ;
};

Management URLMgrMgt: URLMgrImpl {
    // Locate the component which accesses the WWW
    // servers on the local WWW server host
    URLPoller.Host.name == "pukapuka.inrialpes.fr";
}
```

The above example illustrates the use of distribution policies in the *URLNotifier* application. The component in charge of

managing the list of the desired URL (*URLMgr*) should be colocated with the component of the user interface (*ListBox*), because these two components heavily communicate with each other. However, the *URLMgr* component uses a specialized sub-component (named *URLPoller*) for polling URL changes through communications with the WWW servers. We have decided to place the *URLPoller* sub-component on the node which hosts the WWW server of the local environment, in order to reduce significantly the network traffic.

This example includes two kind of policies : one on the component's attributes and one on the sub-components attributes. The second kind of policies is particularly interesting when common or opposite placement criteria (e.g. colocation) are required for a set of components.

However, the combination of both kind of policies draws some problems for their evaluation. Actually, if a sub-component has its own policies while the composite defines new ones, they might contradict each other. For that reason, priority is given to the policies defined for components of the lower levels of the hierarchy. Thus, they are considered before those of the upper levels. In case of inconsistency, the compiler raises a warning and the policy of the upper level is ignored. Let us note that inconsistencies may arise when predicates on a single attribute field are always evaluated to false during resource matching.

3. ARCHITECTURE OF THE CONFIGURATION MACHINE

Once an OCL application is correctly described, the compiler generates the appropriate classes of components, collections and connectors needed by the application. These are placed in a distributed repository. The compiler also generates a script (called the *Configuration Machine script*) that contains orders and guidelines for the initial deployment of the application on a given distributed environment.

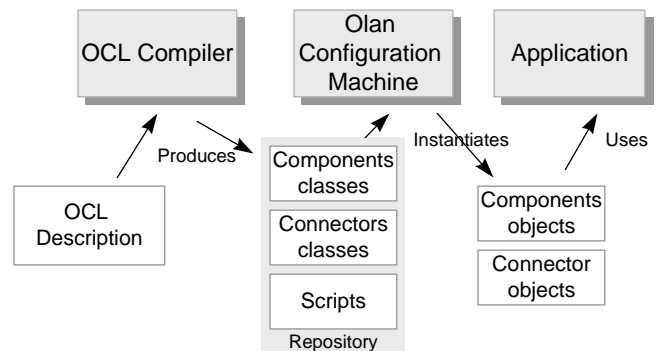


Figure 2. The Olan Development Cycle

The Olan Configuration Machine is in charge of instantiating the appropriate execution structures (components and collections, connectors) in compliance with a given Configuration Machine Script. It relies on two main abstract machines (libraries) : the *Component library* providing features for managing component and collection structures and the *Connector library*, in charge of the connectors. The Configuration Machine itself is

implemented on top of a CORBA-compliant ORB (Xerox ILU[14]) for its internal management of the distribution.

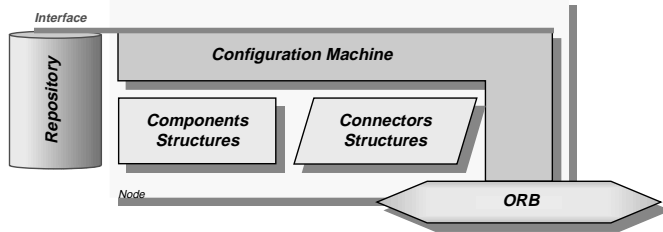


Figure 3. Olan Runtime Architecture

This section presents the basic execution model of components interconnected with connectors. Then we will briefly introduce the role and functions of the configuration Machine, which is in charge of deploying and installing the components and connectors structures on a distributed environment.

3.1 Instantiation of Component Structures

We have already introduced the fact that components correspond to execution structures at runtime. They are composed of integrated software modules and a specific object that enables the communication of the encapsulated code with other components (cf. Figure 1). Since the Olan runtime is implemented in Python, these objects derive from Python classes. The main characteristic of such an object is to be configurable and its purpose is to allow dynamic positioning of the interconnections with other components as well as dynamic loading of integrated software. Figure 4 illustrates the structure of such objects as well as the interconnection principles with the connector objects.

Component object includes two tables, *inCall* and *outCall*, indexed respectively by the name of the provide/react and require/notify services of the interface. Those tables act as binding tables between the wrappers encapsulating the integrated software and the connector objects respectively. Component objects trap every call and redirect it according to the contents of the binding table.

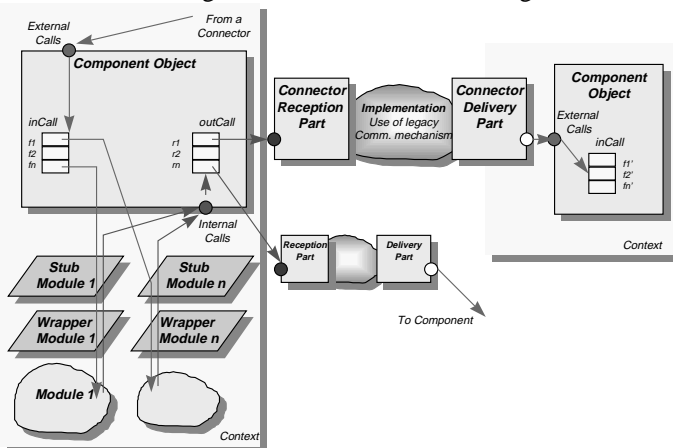


Figure 4. Runtime Interconnection Implementation

Let us notice that a primitive component, as illustrated, is contained in a single context. A context may be a single

process if the encapsulated software can be mapped in one component object context (i.e. can be directly accessed by the Python component object), otherwise it may be needed to have another process that contains the mechanisms to have the component object in Python dialog with its software modules. For example, when trying to have Java classes integrated in a components, we have to launch a Java interpreter from the Python object and have both processes communicate (in our case, UNIX domain sockets are used). A context, in this case, may comprise several processes. When trying to integrate executable software (like a UNIX command), the same principle applies.

3.2 Instantiation of Connector Structures

The main role of connectors is to bind a set of potential senders to a set of potential receivers for communication schemes based on service request or event notification broadcast. A connector is represented by two main kinds of Python objects: the *service adapters* (resp. notification) and the *sender/receivers* objects (c.f. Figure 5).

The *adapters* represent both the entry and exit points for the connector structure. On a sender side, they provide a function that allows a given component to initiate a communication as described in OCL. On a receiver side, they have the ability to call a given function (representing either a provided service or a notification handler), provided by the component. The adapters are also in charge of executing the user-level code which may have been added to the connector description (e.g. code for data flow adaptation).

The *sender/receiver objects* are in charge of :

- encapsulating the use of the communication system,
- handling the possible control flow translations (e.g. when a sender asks for an asynchronous service request while the receivers provide the service in a synchronous way), and
- handling remote communication according to the placement of the interconnected components.

Let us detail the synchronous procedure call connector, which specification is described in Table 2.

Service adapters are first created on each connected components side. Then, if those adapters are on the same context no *sender/receiver object* is created ; both adapters communicate through a method call. In any other case, two objects, the *SCConnSender* and the *SCConnReceiver* are created. In this example, both are ILU objects and they exchange an ILU reference in order to communicate remotely. Then they are linked with their respective *service adapter*.

Information flowing through those objects is already marshaled data (done by the stub at the primitive component level). However, if data transformation code is specified at the OCL level, this code generated by the OCL compiler is inserted in between the sender's *service adapter* and the *sender object*. This code unmarshals parameters, applies data transformation and marshals the new result.

This example is rather simple because the `syncCall` connector handles a 1 to 1 connection. In case of a 1-N connection schema, the *sender/receiver object* structure is somehow more complicated because there must be some treatment for the delivery of the various results to the component that initiated the communication.

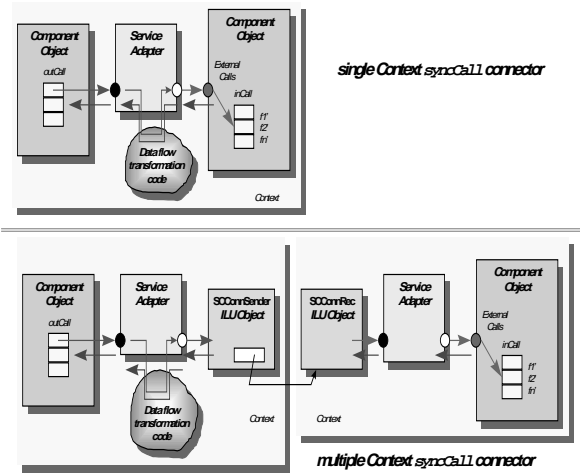


Figure 5. *syncCall* Connector execution Structures

3.3 Deployment of an Application

The deployment and installation of an application is performed by the configuration machine. The initial deployment is performed through the execution of the *Configuration Machine Script* produced by the compiler. This script contains calls to primitives of the configuration machine which main functions are :

- the creation of components and collections,
- the creation of connector instances, and
- the binding between connector and component instances as described in the previous section.

Any application that should be installed is a hierarchy of components whose leaves are primitive components. This hierarchy is derived from the OCL description, and is encoded as a tree. The deployment algorithm is based on a depth first traversal of the tree. For each component, the configuration machine seeks for a node where the component can be installed according to the distribution constraints expressed in OCL. If no assignment is possible, the configuration machine reports the unfeasibility of the deployment to the administrator. Once primitive components are installed, their enclosing components are created; We haven't really detailed the composite component structure but it also corresponds to a runtime structure and basically it looks like a primitive one, except that its bindings are not to stubs or a wrapper, but to the enclosed sub-components. In addition, a composite is replicated on each context of its sub-components. Once the upper level composite component is instantiated, attributes of the sub-components are set and interconnections of the sub-component can be realized. Interconnections require the instantiation of connector

objects that are replicated or not according to the placement of sub-components as described in the previous section.

The execution of the application may then be started through an execution script generated by the compiler. This script allows launching of new applications or the connection to a running application, thus enabling the cooperation of multiple users using the same application.

4. IMPLEMENTATION AND EXPERIMENTS

The implementation of the OCL compiler and the Configuration Machine has been made in Python. Such a language is of a particular interest for rapid prototyping due to the lack of compilation phase, the dynamic typing, reflexivity features, easy manipulation of complex structures such as lists, dictionaries, etc. However, the price to be paid is related to the poor performance of such mechanism. The configuration machine is also implemented in Python and uses ILU, a CORBA compliant Object Request Broker. ILU's purpose is for the internal communication of the configuration between remote nodes. For example, when the machine tries to find a node where a component can be installed, communication between potential hosts are triggered through the use of the ORB. However, ILU is not used for the execution of the application, except if the architecture contains explicit use of a connector that communicate remotely with ILU. The component and connector structures are also implemented in Python, only stubs or wrappers may be partially in Python and in the language of the integrated software modules. Finally, the implementation of the communication inside connectors depends on the kind of connector which is used. For instance, there can be multiple implementations of a remote synchronous call: one using sockets, another one sun RPC, a third one using ILU.

As one can notice, good performances of the runtime system were not aimed at all. The choice of the various implementation languages and tools were motivated by the will to prototype rapidly a complex system, where a compiler and a runtime system are needed. The experience drawn from our experiments mainly concerns the ease of application configuration, the flexibility of the deployment and the transparent utilization of a distributed environment with no particular competence from the programmer.

Besides the examples used throughout the paper and some other applications[2], we have also experimented the Olan environment on an electronic mail application which goal is to help inexperienced users to easily configure some automatic treatments on their incoming messages. For example, filtering functions, forwarding functions, automated move in folders, automatic responders,... are provided in the application as ready-to-use components. Through the OCL language, the configuration of those functions corresponds to the declaration of new components, the positioning of their attributes (such as a filtering expressions) and the binding between components. Distribution is managed transparently, for example, when the user executes its favorite mail browser application on a

different node than its mail server. Software integration has also been experienced, e.g. for the integration of a Netscape browser in a component, for the mail server, ...

The main feedback of the experiments confirms the difficulty to reengineer existing application. First, an application implementation is not always clearly separated in modules, and second, the components are not often designed to work in a distributed environment. The Olan environment is not supposed to answer directly to this problem, however the work of redesigning the application into a set of interconnected components is greatly assisted by the compiler. The communication and the distribution become completely separated from the core implementation, thus allowing non-specialists to build rather complex distributed applications from their own set of components and to produce various distributed configurations of the same application. As a promising consequence, reengineered applications have gained in reliability by using the efficiency of the automatically generated communication and distribution mechanisms.

The second feedback is the necessity of having graphical tools to assist non specialists in using a configuration language. We have always been designing the OCL language with graphical conventions in mind[3], however such a tool is not available today. This is a future direction of work.

The current prototype is built on top of ILU using a Python runtime. The currently available facilities include: an OCL compiler, and a Configuration Machine running on AIX, Solaris and NT platforms.

5. CONCLUSION

This paper has described the Olan Configuration Language (OCL), an architecture definition language for distributed applications design and construction. The idea behind the design of this language relies on the notion of "programming in the large", where the architecture definition of an application is separated from the core implementation of individual software components. By providing a clear separation between communication requirements and components implementation, such an approach brings flexibility in the configuration process of inter-components communication.

Compared to the related works, the main contributions of our proposal are the following.

- The integration of heterogeneous pieces of software through a uniform component-based model.
- The coordination of components through a generic connector model, which allows the application's designer to separate the communication aspects from the components, as well as to specialize some predefined schemes of communications.
- The integration of middleware platform specific features within the abstraction of the connector.
- The specification of some of the dynamic aspects of an application execution, such as the instantiation scheme of

software objects and the dynamic naming of components in a communication.

In addition, an application described with OCL can be derived into an executable image with the help of the OCL compiler. This executable image relies on the Configuration Machine which primary goal is to deploy components on a distributed execution environment, configure the interconnection between components and transparently handle remote communications with the connector objects.

Compared to « traditional » ADLs, the emphasis is placed on three aspects: the integration of existing code, the provision of configurable connectors and the expression of the software components distribution. Code integration is greatly eased by : (1) the use of a type system that brings flexibility when adapting components which are not necessarily designed to work together, (2) the use of complementary information that exports at the interface definition level the encapsulated execution model of components, and (3) by the automated generations of stubs and wrappers that enable the access to the core implementations of components written in different programming languages.

The distribution of components is achieved at the language level with the specification of placement constraints of the application components, which is more flexible than simple static node assignment. At runtime, a deployment algorithm ensures a correct installation of the application according to these constraints.

Future aspects are related with the dynamic reconfiguration of an application, i.e. the way a given configuration may be changed at runtime with the fewest disturbance of the execution as possible. Further work on engineering tools (notably a graphical front-end) are also in progress.

ACKNOWLEDGMENTS

The authors would like to thank discussions with their colleagues of the SIRAC Project. We gratefully acknowledge the CNET for their financial support.

REFERENCES

- [1] Allen R., Garlan D., « Formal Connectors », (CMU-CS-94-115), School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994
- [2] Bellissard L., Ben Atallah S., Kerbrat A., Riveill M., "Component-based Programming and Application Management with Olan", *Object Based Distributed and Parallel Computation Franco-Japan Workshop (OBDPC'95)*, Eds. Briot J.P., Geib J.M., Yonezawa A., Lecture Note on Computer Science, LNCS 1107, Springer Verlag
- [3] Bellissard L., Ben Atallah S., Boyer F., Riveill M., "Distributed Application Configuration", in *Proc. of the 16th IEEE Intn'l Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, April 1996

- [4] Garlan D., and Shaw M., "An Introduction to Software Engineering", in *Advances in Software Engineering and Knowledge Engineering*, Vol. I, Eds. Ambriola and Tortora, World Scientific Publishing Co., 1993
- [5] Kramer J., Magee J. and Sloman M., « Constructing Distributed Systems in Conic », *IEEE Transactions on Software Engineering*, Vol.15 (No.6), pp. 663-675, 1989
- [6] Magee J., Dulay N. and Kramer J., « A Constructive Development Environment for Parallel and Distributed Programs », *Proc. of the Intn'l Workshop on Configurable Distributed Systems*, Pittsburgh, March 1994
- [7] Luckham D. C., Vera J., "An Event-Based Architecture Definition Language", *IEEE Trans. Software Engineering*, vol.SE-21(N.9), September 1995, pp.717-734.
- [8] Object Management Group, « The Common Object Request Broker: Architecture and Specification », Revision 2.0, 1995
- [9] Purtilo J.M., « The POLYLITH Software Bus », *ACM TOPLAS*, Vol.16 (No.1), pp. 151-174, Jan. 1994
- [10] Reiss S., « Connecting Tools Using Message Passing in the FIELD Environment », *IEEE Software*, pp. 57-66, July 1990
- [11] Rogerson D., "Inside COM", Microsoft Press, 1997
- [12] Shaw M., DeLine R., Klein D. V., Ross T. L., Toung D. M., Zelesnik G., "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering*, vol.SE-21(N.4), April 1995, pp. 314-335.
- [13] Vion-Dury J.-Y., Bellissard L., Marangozov V., "A Component Calculus for Modeling the Olan Configuration Language", to appear in *Proc. Of COORDINATION'97*, Berlin, Germany, September 97
- [14] Xerox Co., "ILU 2.0 Reference Manual", Xerox Parc, Palo Alto, CA, July 1996