Distributed Application Configuration

Luc Bellissard[*], Slim Ben Atallah[†], Fabienne Boyer[‡] and Michel Riveill[†]

IMAG-LSR[§]- Projet SIRAC[¶]
BP 53, 38041 Grenoble Cedex 9
email: Luc.Bellisard@imag.fr, Michel.Riveill@univ-savoie.fr

Abstract

This paper presents Olan, a language and a run time support intended to facilitate the design, configuration and evolution of distributed applications made up of heterogeneous software components. Configuration covers two phases: for the application builder, the identification of the software components and the description of their interconnections and communications; for the application administrator and maintainer, the accurate use of system resources provided by the target environment, such as the placement of components on nodes.

The main benefit of Olan is to provide a single unified description of distributed applications, adequate for construction, management, and evolution. The overall description is independent from the components' implementation, so that the configuration process, e.g. the production of different versions of an implementation, is decoupled from the programming process.

The features of the configuration language and the functions of the runtime support system are illustrated through the example of a distributed teleconferencing application.

1 Introduction

Distributed software systems are usually composed of a number of mutually interacting software components. We focus our work on applications that involve several users working on a set of workstations, sharing information and interacting in real time. Examples of such applications are teleconferencing, electronic diaries, software development environments, and workflow applications.

The *construction* of an application refers to the description of the software entities involved, the definition of their mutual interactions and the evolution of entities and interactions throughout the lifetime of the application. *Management* deals with the accurate use of system resources according to the application's requirements, (e.g. placement of components on a distributed network, observation, access control) or the correct setting of parameters necessary to produce an executable image of the application (e.g. the number of simultaneous users in the application).

These applications are conveniently described, constructed and managed in terms of their global software architecture. However, the current techniques available on distributed platforms (such as remote procedure call, multicast, client-server model) hardly help in the global design process of decomposing an application into a set of interacting components. They also ignore management aspects such as the ones stated before. Runtime support is usually missing for the construction and the management of applications that exhibit a structure which is any more complex than a simple client-server arrangement.

Module Interconnection Languages are a solution to handle both construction and management, collectively referred to as *configuration*. They isolate the program structure from the context of execution, allowing the developer to concentrate on the functional requirements of software modules. MILs are higher level languages in which the interconnections between components are described independently of the deployment platform. A runtime support encapsulates communication between modules as well as data transformation, so that the handling of interfacing requirements is decoupled from that of functional requirements. Programmers are freed from the constraints imposed by the underlying architecture, the language processors, or the communication media. Furthermore, once an application has been configured for one execution environment, then the deployment for another execution environment is treated separately, automatically and independently of the implementations of

[*] Institut National Polytechnique, Grenoble

[†] Université de Savoie, Chambéry

[‡] Université Joseph Fourier, Grenoble

[§] IMAG-LSR is a joint laboratory of Centre National de la Recherche Scientifique (CNRS), Institut National Polytechnique de Grenoble, and Université Joseph Fourier, Grenoble

[¶] SIRAC is a joint project of IMAG and INRIA modules. *Polylith* [8] is an example of such a language and runtime. *Conic* [6]and more recently *Darwin* [7]extend the approach proposed by *Polylith* by enabling the designer to describe an application's structure as a hierarchy of interconnected components instead of a flattened collection of modules. Our work [4] extends the proposals of Darwin with a language that addresses the following issues:

- encapsulation of pre-existing or custom-made software entities, whose granularity may vary,
- expression of *interactions* between software entities, in terms of functional dependencies and communication types,
- decoupling of interactions from component's implementation and construction of complex interactions with the introduction of *connectors* [9],
- dynamic evolution of the application structure and the interactions, and
- management of distributed applications with mechanisms that allow placement of entities, monitoring of executions, etc.

Section 2 presents a distributed application scenario used as a testbed for the language and the runtime. Section 3 describes the concepts of the Olan configuration language. The run-time environment for our model is presented in section 4 and the result of our experiment is described in section 5.

2 CoopScan: An Example of a Distributed Application

This section describes a simple application scenario which is later used to illustrate the capabilities of the proposed language. This application, named CoopScan [2], aims at providing a support for teleconferencing between distributed workers, who interact through an audio channel and shared documents. A complete description of the application may be found in [4]. For simplicity reasons, we only describe a simple scenario restricted to document sharing. Various types of documents are available through applications initially designed for single-user usage, such as XFig, XEmacs, Grif, etc. CoopScan provides functions to enable WYSIWIS interactions (What You See Is What I See). It also allows dynamic participation of conferees and manages roles and access rights.

Basically, a user willing to enter a cooperation launches a *Session*, that gives him access to the shared documents. A *Session* is made of two kinds of entities: the *Application*

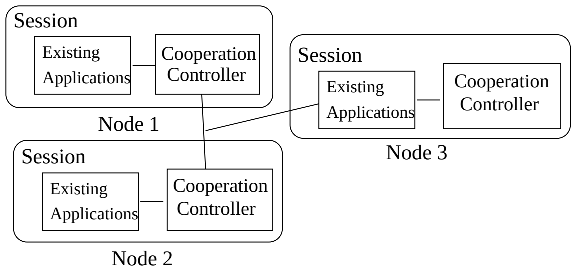

Figure 1. CoopScan general architecture

that performs actions on the shared documents, and a *Cooperation Controller* that manages the cooperation. *Sessions* are usually located on distinct nodes (cf. Fig. 1).

The cooperation works as follows: an action performed on a document by an application is first passed on to the local *Controller*, which checks if the action is allowed to the user. If so, the action is actually carried out onto the local user interface and passed on to the remote *Controllers* that ask their corresponding application instance to replay the action in the same way. This is how WYSIWIS is ensured. More details on the *Controller*'s roles may be found in [2]. At a finer level, the *Controller* is made of two components: a *LocalAgent* and a set of *DistantAgents*. The role of the *Local Agent* is to receive actions from the local applications, to validate them and to broadcast actions onto the remote nodes. The role of the *DistantAgent* is to receive actions from remote nodes and to ask the local application for their local execution. There is a *DistantAgent* associated with each of the remote users. Fig. 2 shows a communication path between *user1* application to the remote application (here belonging to *user2*). Clearly, the action performed by *user1* is passed on to its *LocalAgent* and then broadcast to the remote applications by the way of its associated *DistantAgent*.

The application was initially developed on Unix using custom-made collaborative protocols built on sockets (used as communication tools). It has been reengineered using the Olan language described in the following section.

3 The Olan Component-based Language

An application whose configuration is described with Olan can be viewed as a hierarchy of interconnected **components**. Each level of the hierarchy is a separate Olan description derived from **a component class** which encapsulates components - in fact components' instances - contained in the next level. The leaves of the hierarchy are components with no Olan description, which are called **primitive components** and encapsulate 'real' pieces of software, like C++ objects or C modules. They are derived from a **primitive component class**. We have introduced the concepts

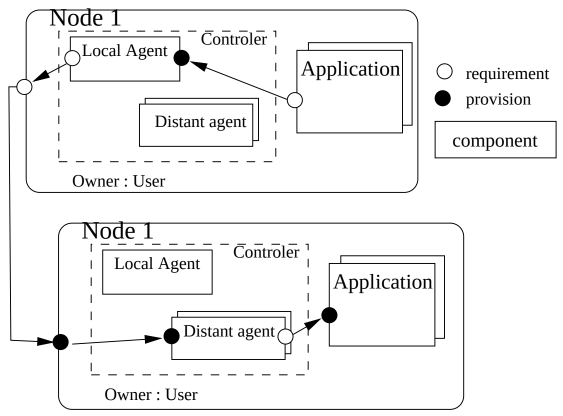

Figure 2. A communication example in CoopScan

of class and instance in a Module Interconnection Language to emphasize the distinction between the unit of reusability and encapsulation (the component class) and the effective unit instantiated at run time (the component).

3.1 Components

As in object oriented programming languages, components are described by an interface and an implementation.

3.1.1 Interfaces

The interface of a component class contains information that allows using the component without knowledge of its implementation. We have extended the usual concept of an interface, as defined for example in the CORBA IDL, to include the description of all the properties required from a component to work properly. An interface contains *services*, *notifications* and *attributes*.

Services are functions or procedures that can be executed or requested by a component. *Provided services* are always offered by the implementation of a component class while *required services* are the services expected to be called by the implementation of the component class.

Notifications are events whose broadcasting can trigger the execution of reactions on several components. A component which receives a notification may optionally react to it by executing a code sequence called a **Reaction**.

Notifications are not necessarily connected to a reaction, nor is a reaction necessarily triggered when it receives a notification. This is the intrinsic property that distinguishes services from notifications.

Attributes are typed variables whose values are set at instantiation time and can change during execution. The value of a given attribute can be imported from the implementation or set from outside the component.

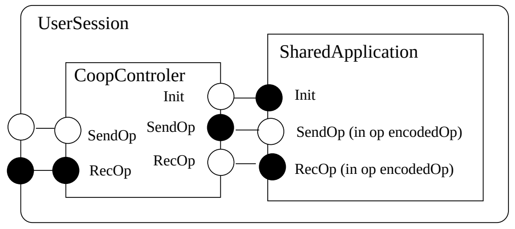

Figure 3. A composite component

3.1.2 Implementations

A component class implementation is either made of a hierarchy of "sub-components"or of "real" pieces of software. We thus distinguish two kinds of components.

A **primitive component** class is the unit of encapsulation and reuse of existing code. The implementation of a primitive component class is made of pieces of software (C modules, C++ classes,...) that implement the provided services declared in the interface, using the required services.

A **composite component** class, is the main structuring tool. Its implementation is made of "sub-components" and interactions between them. "Sub-components" are instances of component classes described elsewhere. Compared to usual programming languages, the implementation defines which other components are needed and how they should communicate to realize the interface rather than how the functions of the interface are to be directly implemented.

Still referring to the teleconferencing application, Fig. 3 shows a component class that contains a *Controller*, which filters actions coming in and out of the *SharedApplication* and transmits the allowed actions to its "super-component". Roughly, this class represents a user in a collaborative session of the application. The horizontal connecting lines represent connectors, which are described in the next section.

3.2 Connectors

Connectors are the units that mediate the interactions between components. They establish the rules that drive component interactions and specify the required protocol at runtime level. Rules at the connector level include: the interface conformity rules (e.g. parameter types conformity, homogeneity of connections,...), the protocols used for the effective communication (asynchronous vs. synchronous communication, event broadcast vs. remote procedure call,...), and the specification of an expected behavior, such as a set of constraints expressing Quality of Service requirements (e.g. for multimedia communication support).

The connector description needs to define:

- the *kind* of connector, that distinguishes those used

Connector Name	Source(s)	Destination(s)	Specification
methodCall	1 required service	[1..n] provided service	synchronous method call on 1 randomly chosen component, parameter compatibility
broadcastMethodCall	1 required service	[1..n] provided service	synchronous method call, sequentially realized
broadcastEvent	1 notification	[0..n] reactions	asynchronous event broadcast
createInCollection	1 required service	1 provided service of a collection	creation of a component of the collection then synchronous method call on this component

Figure 4. Available connectors for inter-component communication

for interconnection purpose and those used for the mapping of the interface to an implementation,

- the allowed *source* of the communication,
- the allowed *destination(s)* of the communication, and
- the *specification* of the expected behavior, the constraints and the protocol.

The concept of a connector has already been introduced in the literature. In [1], connectors are specified using a subset of the CSP process algebra to declare the expected behavior and the associated constraints. This, combined with the inclusion of some CSP specifications in the interface of components, allows compatibility checking between connectors and components to be performed.

The Olan platform for now supports several kind of connectors. Their rules are still very simple and do not yet integrate constraint specifications. Currently, there is no language or formalism to express those connectors ; they are build-in connectors provided by the runtime support and available in the Olan language. The following table (Fig. 4) briefly describes some available connectors.

Interactions are the effective communications that take place between a set of components. They are issued from bindings between sub-components services or notifications/reactions, using connectors that describe how the interaction should work. Interactions may thus be seen as "the instances" of a connector in a well-defined context, i.e. in a specific implementation with its sources and destinations specified.

The **mapping** to the interface is the process of binding the interface requirements and provisions of the implementation. "Lines" drawn between a service - or notification/reaction - and a sub-component, may be seen at execution time as a communication link between the enclosing component and its sub-components. That is why we use connectors here as well, as a means of building those links.

The component described in Fig. 3 may now be completed by the description of both the interactions and the mapping to the interface. Fig. 5 shows the Olan syntax for

```
component class UserSession {
interface
  require SendOp (in operation);
  provide ReceiveOp (in operation);
  ...
implementation
  theCont = inst CoopControler;
  theAppl = inst SharedAppl;

// mapping using methodcall
// connector
  ReceiveOp => to theCont.ReceiveOp;
  theCont.SendOp => SendOp;

// interaction
  theCont.Launch => theAppl.Init;
  theCont.SendOp => theAppl.ReceiveOp;
  theAppl.SendOp => theCont.ReceiveOp;
};
```

Figure 5. Interaction and Mapping example of declaration

the use of connectors. The keyword **inst** is the declaration of sub-components.

4 The Olan Framework

The Olan configuration language is part of a framework that provides tools to build and maintain distributed applications. The framework is divided into two categories of tools:

- *Tools* for programming (Visual Programming Environment and compiler) and management (AdminTool), and
- *Execution* units that are either generic (runtime support) for all applications or specific (executable image of one application).

In the rest of this section, we focus on the language facilities and the runtime support architecture that enable the production and the management of an application.

4.1 Using Olan to Build Applications

Building an application with Olan requires first a configuration description expressed with the language presented in section 3. The compiler then produces an executable image that can work on the runtime support. However, the language only catches the static aspects of the application, while an execution is obviously dynamic. In the static description, components are predefined, interactions are set up once for ever, etc. Clearly, this is not satisfactory especially for the types of applications we aim at constructing. There is a need to allow dynamic instantiation of components, to create sets of components and to have destinations of interactions and mapping named according to dynamic properties..

Instantiation is the act of describing the set of sub-components in an implementation. There are several ways of instantiating a component, according to the time when it should be effectively created at run time. The keyword **inst** refers to a predefined instance that is created together with the enclosing component. To express more dynamics, a component can be predefined in the class implementation but is actually created at the first invocation of one of its services. This is the *lazy instantiation* scheme of Darwin.

```
// implementation of UserSession
theCont = dyn inst CoopControler;
```

In the *UserSession* component (Fig. 3), the *Controller* is dynamically instantiated, as there is no need to instantiate it before the user decides to enter the Session. So the component is created the first time a service of *UserSession* is activated.

Collections are introduced to facilitate handling of multiple components of a same class. A collection is a set of components whose number at execution time lies within a range defined by a minimum and a maximum cardinality. Collections allow the easy manipulation of a set of entities which can evolve dynamically and are accessed through a single interface.

```
// implementation of UserSession
theApplSet = collection [0..n]
   of SharedAppl;
```

In order to allow multiple applications to be 'shared' in a same Session (e.g. an Xfig and an Editor), we group them in a collection as shown in the previous Olan lines.

As collections are the keystone of dynamic instantiation, the predefined services **Create** and **Delete** are provided in any collection. In addition, a special connector can operate as a creator of a new component of a collection before accessing the specified service.

```
// implementation of UserSession,
// definition of the interactions
theCont.Launch
  => theApplSet.Init
  using createInCollection;
```

In the implementation of the *UserSession* component, when the *Controller* asks for the application to be launched, this connector induces the runtime to first create a new component in the collection and then activate the *Init* service of this particular component.

The introduction of collections raises new issues concerning the *naming of components.* In a distributed system, objects are generally named by their object identifier (Oid). This naming scheme seems somehow restrictive and incompatible with collections, when the destination of a communication is a priori not known. Therefore, we use associative naming facilities to specify the destination of an interaction (or a mapping). Associative naming is based on the existence and the run time value of attributes. Among these attributes, one can always find a unique object identifier, so a direct naming is still possible. However, in the case of collections, it is often more useful to define the interaction's destination(s) with the help of more general and semantically richer attributes.

```
// implementation of UserSession,
// definition of interactions
// with associative naming
theCont.SendRemoteOp(operation, remoteApplId)
  => theApplSet.ReceiveOp(operation)
  where theApplSet.ApplId = RemoteApplId
 using methodCal;
```

Still referring to the *UserSession* component, the *Controller* which receives a remote action (from another possibly remote *Session*) knows to what kind of application it should transmit it but not to what particular component in the collection. For this reason, the *Controller* transmits the action to the *SharedApplication* whose *ApplId* attribute fits with the identity of the target application of the action.

4.2 Runtime Support

The runtime support is the foundation of the framework and the basis for the execution of a component-based application. It provides facilities for handling components, connectors, management policies, etc. These facilities include:

- knowledge of the application structure and configuration, in terms of components,
- RPC and asynchronous distributed events facilities, needed for connectors,
- collection handling, that provides dynamic evolution of the component hierarchy,
- associative naming facilities,
- complex connector handling, and
- management instrumentation and policies interpretation facilities.

As the runtime environment foundation, we have chosen Oode [5], an object-based distributed platform developed by Bull, for its rapid prototyping facilities. Indeed, Oode provides a concurrent object-oriented language (an extension of C++) that allows transparent programming of the distribution and the manipulation of shared objects with concurrent access control mechanisms. Such facilities are convenient for a first experiment since they ensure easy communication and information sharing.

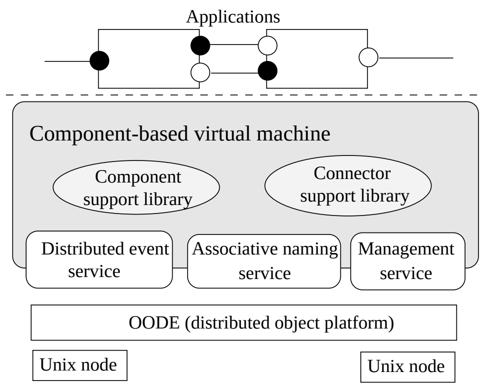
Applications
Component-based virtual machine
Component support library
Connector support library
Distributed event service
Associative naming service
Management service
OODE (distributed object platform)
Unix node
Unix node

Figure 6. OLAN Runtime Support

The key decision of our runtime implementation is to provide an object for each entity described with Olan. In other words, components, connectors, collections,... exist as distributed objects. The runtime system is therefore a virtual machine for component-based programming. The machine is composed of a set of libraries that represent components, connectors, collections, etc.. These libraries are implemented as a hierarchy of classes. The components that are specific to the application are created by inheritance from the *component support library*; a similar scheme is used for connectors.

The component-based virtual machine needs additional services that are not directly provided by Oode:

- the distributed event service,
- the associative naming service, that enables a set of objects to be named through their properties i.e. static properties such as the class name, an attribute value or dynamic ones like the value of an attribute,
- the management service, that offers ways to instrument the application with sensors and actuators and handles the interpretation of policies.

The Oode platform itself is built on top of Unix (AIX and Solaris 2) and DCE RPCs and threads. In a further step, we may decide to consider other kinds of platforms as the basis of the Olan environment, such as a CORBA platform or a software bus like Polylith.

5 Experiments and Preliminary Results

The CoopScan application (cf. section 2) was already available on Unix; communication between nodes was achieved with TCP or/and UDP sockets. Recently, we have started an experiment aiming at reengineering this application. The first step of our experiment was to re-implement the application on top of Oode in order to appreciate the impact of object sharing and transparent distribution. The second step of the experiment was to redesign the application as an interconnection of components, with a major constraint: reuse the existing code as much as possible [4].

This experiment has highlighted a number of benefits. The programming process of an application is much simpler. The programmer can focus on the implementation of components with no consideration for the platform and the overall architecture, whereas the architect can assemble components with no additional implementation work. For example, communications are simply specified with an appropriate connector and the runtime uses the adequate system mechanism supplied by the platform. We are no longer concerned with communications mechanisms such as RPC. The architecture of the application is also much more clearly exhibited by the structuring in terms of components and by the higher level description of communications with connectors. The maintenance of the application is now clearly independent from the implementation of software modules, thus making evolution of new versions quicker and more flexible. For instance, we are able to produce different versions of CoopScan, one with limited number of users, one with a shared editor and the other with XFig and the editor.

The problems we have faced are related to the execution scheme of the application, i.e. the description of the application in terms of cooperating processes. We still have no description of those aspects in our language, which makes harder the integration of existing components. In Durra [3], the philosophy was to describe the application in a more 'task-oriented' way; we plan to consider the integration of such features in our approach, which may be considered at this stage as more 'software-oriented'. Another problem is the extra cost due to the runtime support. The runtime has a very simple design, in which every entity of the Olan language is mapped on an Oode object. This is because we wished to build rapidly a prototype of the runtime in order to test the program construction facilities. Efforts are being done to redesign the runtime system for better performance. Finally, the management features have not yet been implemented, besides simple facilities for the placement on a distributed network. We are still in the process of designing the management service and its relation with the language.

6 Conclusion

This paper has described a framework for the construction and configuration of distributed applications. In our view, distributed applications are constructed by assembling a set of interacting software components, some of which are pre-existing applications

We are considering an approach which combines object-oriented programming for the description of individual software components and system integration techniques for the description of relationships and interactions between components. The proposed framework includes a model and a language which allow a high-level specification of involved components, and a run-time support for the actual execution and operational control of the application.

The main features of the model are :

- *a hierarchy of software components*. Primitive components encapsulate a piece of software written in any programming language (including pre-existing applications). Compound components are made of a number of interacting components. Components are instances of component classes.

- *component classes* are described by their interface, which provides information about the provided services, as in usual object models, but also the services required from outside and the reactions to incoming event notifications. In addition, an interface includes a number of public attributes used for overall management and control. Interfaces are built by extending the CORBA IDL notation with additional features.

- *connectors* are the units of communication mediation between components. A given connector type specifies the component types to be bound as well as the communication protocol between them. Using connectors allows a clear separation between the code of each component and the specification of the interactions between them.

- the model also allows instantiation of components, management of collections of components, and associative naming of components (using their attributes). These facilities provide support for the control of the dynamic aspects of an application.

At the current stage, the Olan framework is implemented on top of a distributed object oriented platform available in Unix environments. This implementation strategy is expected to allow rapid prototyping of the environment. The event service and the associative naming service are already operational, while the management service is under development. Work on the compiler of the Olan configuration language has not yet begun, but an intermediate representation that produces an executable image of an application is available. A first operational prototype of the whole environment is expected by mid-96. In parallel, several real-world applications are being designed using the model and will be demonstrated on the prototype. This should allow us to assess the benefits of the model for rapid application design and the extra cost of using the Olan run-time system.

Future work includes : studies on a formalism for the specification of connectors; extension of the framework for the specification of instrumentation and management policies; provision of development tools based on a 3-D visual programming interface. Furthermore the overall framework will be ported on various host platforms (e.g. CORBA environment).

Acknowledgments This work was partially supported by the European ESPRIT project Broadcast and by the CNET (France Telecom). We would like to thank Roland Balter and Sacha Krakowiak for their careful comments, and the Sirac Team, especially Emmanuel Lenormand, Vladimir Marangozov and Jean-Yves Vion-Dury, as well as André Freyssinet and Serge Lacourte from Bull Research Center / Open Software Systems.

References

[1] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

[2] R. Balter, S. Ben Atallah, and R. Kanawati. Architecture for synchronous groupware application development. In *Proceedings of the Sixth International Conference on Human-Computer Interactions (HCI International '95)*, pages 371–379, Tokyo Japan, 9-14 July 1995. Elsevier.

[3] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: a structure description language for developing distributed applications. *IEEE Software Engineering Journal*, Vol.8(No.2):83–94, March 1993.

[4] L. Bellissard, S. Ben Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with olan. In *in Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, Tokyo Japan, June 21-23, 1995. to appear in LNCS Springer Verlag.

[5] Bull Open Software Systems. Oode: Une plate-forme objets pour les applications coopératives. In AFCET, editor, *AFCET*, Paris - France, Novembre 1994.

[6] J. Kramer, J. Magee, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, Vol.15(No.6):663–675, 1989.

[7] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, Pittsburgh, March 1994.

[8] J. M. Purtilo. The polylith software bus. *ACM TOPLAS*, Vol.16(No.1):151–174, Jan. 1994.

[9] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *in Proceedings of Workshop on Studies of Software Design*. LNCS Springer-Verlag, 1994.