Controllers: Reusable Wrappers to Adapt Software Components ⋆

José M. Troya and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga. Campus de Teatinos. 29071 Málaga. Spain.

Abstract

This paper discusses the modular development of software applications in Open Systems using reusable components and controllers. In this work a component model for open system is proposed, in which components encapsulate computation, systems deal with the creation and communication of components, and the rest of the context-specific requirements and concerns are implemented by first-class, reflective, reusable entities, called *controllers*. Controllers are thought to be the natural evolution of the traditional object wrappers or filters to the realm of component-oriented programming, that can be added to components in a modular way, modifying their behavior according to the requirements they implement.

Key words: Open systems, software components, middleware platforms, reflection.

1 Introduction

The increasing use of Open and Distributed Systems (ODS) for the development of applications, together with the emerging need of a global component marketplace, are changing the way software is developed nowadays. Reusability and late composition are two driving forces towards the separation of the computational and interoperational aspects of components, while ODS-specific requirements have to be incorporated into the user application too, and in an orderly manner.

Current architectural approaches to deal with these issues rely on components and connectors. Components encapsulate computation, while connectors describe how components are integrated into the architecture. Although this

⋆ Work supported by the Spanish CICYT project TIC99-1083-C02-01.

separation of concerns has clear advantages for system design, verification and reuse, it also presents some limitations. Connectors are good for defining and managing the interconnections between components, but they still lack the ability to abstract other important *properties*, like resource discovery and management, placement policies, reliability features, and other context-specific requirements [2].

On the other hand, the approach followed by the main commercial component platforms (such as CORBA, DCOM or JavaBeans) to cope with the ODS specific requirements is to define and implement new services and features, extending and complementing their basic models. Examples are the new CORBA services and features [4,21], the Microsoft Distributed Component Architecture (MDCA) and its services (security, transactions, messages and clustering) [17], and the new versions and additions to JavaBeans (Glasgow, Enterprise, BeanConnect and InfoBus) [www.javasoft.com]. However, these extensions do not provide a good solution in the long term, since they may hinder the reusability and portability of components. For instance, it is difficult to migrate to a different environment a CORBA component that strongly relies on a particular CORBAservice. Even worse, specific requirements not covered by the model should be undertaken by the components themselves, which unnecessarily complicate their design and development, hindering again their reusability, portability and openness. Another problem with model extensions is that the models were originally designed without any of them in mind, which tends to produce hybrid and unnatural results. For example, the asynchronous messages defined in MDCA constitute quite an unnatural communication mechanism for COM components, that traditionally use RPCs.

A reflective approach

A more appropriate approach for application development in ODS considers components as black boxes that transparently modify their behavior through *controllers* —also called layers, home processes, meta-actors or filters in other models—, first-class entities that modify the component behavior by wrapping them, taking care of the context-specific concerns and requirements of components [1,3,5,11,14,15,23]. However, those meta-components suffer from several limitations that restrict their wide use in component-based software development:

(1) They all lack of a common and uniform structure.
(2) Most of them are just computational filters with delay capabilities at most, even without state.
(3) They are defined in terms of the objects they are attached to.
(4) They are not reusable.

(5) Their functionality and behavior is too tightly coupled to the components they wrap.
(6) Finally, their composition is usually undefined.

Our proposal is also based on a reflective approach and uses wrappers to adapt third-party components to the context-specific requirements, but tries to overcome all those limitations. In the first place, our controllers share the same structure, and they are not mere computational filters: they not only capture and modify messages, but they can also split, reorder or join them, reply to messages, or even interrogate their environment and reconfigure themselves accordingly. Second, controllers are designed to be commercial off-the-shelf (COTS) reusable entities, and independently defined from the components they will be later attached to. And finally, they can be composed to simultaneously enforce several requirements to a given component.

Based on this reflective approach our model offers a three-layered structure: "Systems-Controllers-Components". Systems can be simplified to the minimum, offering just the infrastructure for the creation and communication of components; components encapsulate computation; and the standard add-on controllers provide components with the required behavior, according to the user requirements.

The idea is to ease the task of building applications by using COTS components and controllers, in a software marketplace with room not only for systems and components manufacturers, but also for developers of reusable controllers. In order to do so, some goals must be achieved. First, components and controllers should be defined in such way that controllers can be added to components in a compatible, modular and independent manner, and composed to apply multiple properties simultaneously to a component. Secondly, formal methods and models are needed for specifying the behavior of the components, the controllers and the aggregates, for reasoning about them, and for proving that the application requirements can be met when putting all the pieces together. And finally, interoperability mechanisms are needed for integrating components from different models, making them interoperate seamlessly, and for incorporating legacy applications to our systems.

This paper presents a new component model for open systems that tries to meet all those requirements. It defines the concepts of components and controllers, allows their modular composition and aggregation to build applications, and it is devised to address in an independent manner many of the ODS specific issues, such as heterogeneity, component evolution and dissemination, dynamic reconfiguration, or environment-awareness. The model offers two different parts. First, a communication infrastructure based on asynchronous messages with local broadcasting capabilities. And second, a reflective architecture on top of it to wrap components with controllers. The model is also

supported by a formal framework in Object-Z for specifying its concepts and providing reasoning mechanisms about the components [20].

The structure of this document is as follows. The next section describes the basic concepts of the model and its communication mechanisms. Section 3 describes the reflective mechanisms and the basic properties, together with an example that shows how the model can be used to build up an application from existing components and controllers. Section 4 introduces an interface description language, and section 5 discusses other interesting aspects of the model, like its interoperability with the existing component platforms, or some of its limitations. Finally, we relate our contribution to other works in this area and draw some conclusions.

2 The SC Component Model

In the first place, the model provides components with a set of communication mechanisms for interoperating among themselves, together with reflective mechanisms to modify their behavior according to the application requirements (by means of controllers). This section describes the model communication mechanisms, while its reflective features are covered in section 3. The name **SC** comes from *Self-Coordination*, a concept that claims that each component should be responsible for achieving its own goal, and that constitutes one of the underlying ideas of the model.

In **SC** communication is based on mailboxes and asynchronous message passing, each component having a mailbox where other components can send messages to. Mailboxes have a unique global address, that must be specified when delivering messages to them. On top of that, we have also added *inspection* and *local broadcasting*, two facilities that allow components to cope with both the static and dynamic requirements of information passing in ODS. Inspection is used to interrogate components about the methods they implement, and local broadcasting allows components to send messages to all components currently at a domain (a set of related machines). In the following sections we will discuss all those concepts in more detail.

2.1 Components

Generally speaking, any computational entity can be modeled as an object (although it may be implemented by many), with a state (given by the values of its attributes) and some access operations (its methods). We will refer to a *component* as an object encapsulated with an interface compatible with the

communication mechanisms offered by the system. The capsule abstracts its behavior, hides its implementation and allows its interaction with other components. Besides, components need to be able to be independently deployed and subject to late composition by third parties if we want to use them in a global component marketplace [18].

2.2 Messages and Mailboxes

In **SC** components interoperate using asynchronous messages, a very appropriate mechanism to express and implement the communication among components that takes place in ODS. In order to minimize the system requirements and to model the real requirements of large ODS, we will not suppose that transmission times are bounded, that the relative order between messages is preserved, or that transmissions are error-free. As we shall see, these issues can be separately addressed by the reusable controllers.

Messages are information entities with a header and a body. The header is a set of fields with the delivery information (destination and originator mailbox addresses, message reference, timestamp, the result of the operation for reply messages, etc.). The body is just another field (`Info`) with no predefined structure used to store the data being delivered with the message (operation parameters, etc.). Then, the structure of messages can be described as follows:

```
String    To;        // Target mailbox
String    Reply;     // Source mailbox
String    Subj;      // Message selector
int       Ref;       // Message reference
int       RefTo;     // Reply reference
int       Result     // Result of the operation
Date      Sent;      // Timestamp
Date      ReplyBy;   // Reply deadline
int       Cid;       // Controller id
String    Info;      // Message data
```

An important field is the message selector (i.e. the *subject*) that determines the operation to be executed by the target component. For every method f implemented by a component, we define four different message selectors: $!f$, $?f$, Re:$!f$ and Re:$?f$. The first one invokes the method, and Re:$!f$ is used for replying to it. Selector $?f$ asks the destination component whether it implements method f or not, and Re:$?f$ answers this question. Besides, message selector "??" queries a component for the methods it implements, that is, for its complete interface.

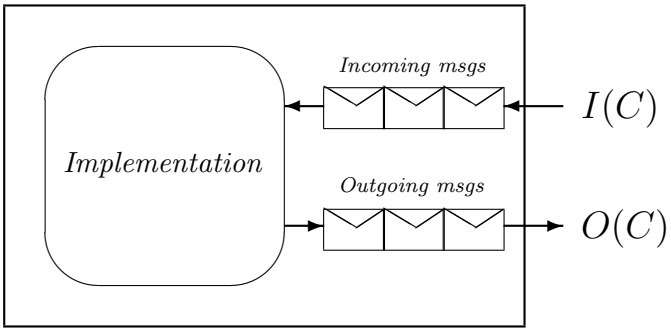

Fig. 1. A component C and its mailbox.

Each component has a *mailbox* through which the component sends messages to other mailboxes and receive messages from other components. Mailboxes have a unique global address and two FIFO queues, one of incoming messages and other of outgoing messages (fig. 1). Mailboxes offer two basic operations: `Send` to send messages to other mailboxes, and `Receive` to read a message from the mailbox incoming queue. `Receive` is a blocking operation if the queue is empty. When created, a mailbox is registered at the current machine and assigned a unique global identifier (its address), that has two parts: a local name and the machine address (e.g. `av@lcc.uma.es`). The sender of a message can specify just a destination name, meaning that the mailbox belongs to its local machine, or a complete mailbox address. But it can also specify a special name (`BCST`) so that the message gets sent to all mailboxes currently at the destination domain.

2.3 Domains

In our context, a *domain* is a set of interconnected machines that defines the *environment* of a component. Domains are structured as trees, where the nodes are machines and the existence of an arc from machine M_1 to its child M_2 means that M_2 wants to receive all broadcast messages sent to M_1. Each domain is uniquely identified by its root address, and subtrees of a domain are also domains.

Local broadcasting to domains is a powerful mechanism since it permits to deal with many ODS-related issues in a simple way, like environment-awareness, dynamic monitoring, adaptation of activities or resource discovery. And as we shall see, it also helps alleviating one of the drawbacks of mailboxes: having to specify the destination address when sending a message.

2.4 Interfaces

Components being black boxes, their behavior is defined by their interfaces. In general terms, the *interface* of a component C is given by two sets: $O(C)$ and $I(C)$. The set $O(C)$ is the set of message selectors that the component sends out (its outputs), while $I(C)$ is the set of message selectors that the component receives and implements (its inputs). Received messages not understood by a component are discarded.

Traditional object-oriented interfaces contain only information about the incoming messages. However, in a component-based model it is important to consider the outgoing messages too. Without them it is not possible for instance to check the substitutability between two components, since we do not only need to know the services a component implements, but also the services it requests to other components.

Based on these two sets it is possible to define the concepts of (syntactic) *compatibility*, *replaceability* and *equivalence* of components. Two components are said to be equivalent ($A \cong B$) if the sets that define their interfaces are the same; compatible if all messages interchanged between them are understood by each other; and we will say that component C is replaceable by component D (or that D is backwards-compatible with C) if D implements all messages that C does, and D's answers to them are the same as C's (a more detailed description of all these concepts can be found in [20]). There are different degrees of replaceability, depending on whether the new component can substitute the old one in every possible application (*strong*), only in particular ones (*relative*), or whether additional components need to be incorporated to the application when replacing the component (*weak replacement*). $\cong$ is an equivalence relation, and therefore introduces equivalence classes. In the following, $[C]$ will denote the representative component of the equivalence class of components whose interface is the same as C.

2.5 Applications

Components are not self-sufficient, they need the services and resources offered by other components to achieve their goals. Therefore the concept of *application* is introduced as a parallel composition of components that interact to accomplish a goal. The goal of the application usually coincides with the goal of one of its components, that we have called the application *initiator*.

More formally, we can define an *application* A as a parallel composition of equivalent classes of components (modulo $\cong$) $A = [C_1] \parallel [C_2] \parallel \ldots \parallel [C_n]$, where each one is compatible with at least one other. By using equivalence

classes of components we can abstract from their particular implementations. In this way, applications can be designed with some degrees of freedom, needed to model the dynamic behavior of open systems. For instance, this definition allows components to evolve or even to be replaced by other equivalent components.

In our notation it is also possible to indicate the existence of more than one component of the same type, as it happens in the case of an application with one producer and two consumers. In general we denote by $[C]^m$ the parallel composition of m components equivalent to C. Each of them will have a different identifier when executed, i.e. each process will have a distinct mailbox address. This notation can be extended to indicate a *variable* number of components of the same type. We denote by $[C]^*$ the parallel composition of a indeterminate number of components equivalent to C, a number that can even change over time. An example of an application where this happens is shown later.

Applications can be also composed to build up another applications. The parallel composition operator "$\|$" can be extended for application composition, where it is an associative and commutative operator.

Given an application, it is important to know the sets of components needed to carry it out. In that sense we say that an application A is *closed* if all methods requested by its constituent components are implemented by other components in the application. With this, a *closure* $\mathbf{C}(A)$ of an application $A = [C_1] \parallel \ldots \parallel [C_n]$ in a domain d is the parallel composition of its components together with the minimum set of components $D_1, \ldots, D_m$ available in d, such that application $\mathbf{C}(A) = [C_1] \parallel \ldots \parallel [C_n] \parallel [D_1] \parallel \ldots \parallel [D_m]$ is closed. By definition, the closure of an application does not have to be unique for a given domain, although this is not an issue at this point.

In order to execute an application we need to close it first, and therefore add to it the extra components required (if any). In general, these extra components will be taken from the existing ones in the domain where the application is run, and this is the reason why the closure of an application depends on the domain. In this way applications are built incrementally and from existing components.

It is also important to note that the closure of an application coincides with its initiator's closure. This is one of the key aspects of our model, whereby applications can be defined by their initiators which, when executed in a domain, 'pull along' with them the set of required components to build up the whole application. This fact, together with the ability to find them and adapt to their interfaces, constitute the basis of the idea of *Self-Coordination* of components, that has given the name to the model.

2.6 Implementation

As a general model, the **SC** mechanisms were designed independently from any specific language or platform, and can be added to any of them as a library, set of classes, packages, or whichever natural importation mechanism is available for that particular language.

A first prototype of the model has been implemented in Java, where the model is available as a set of packages. The implementation is based just on the core Java classes, and uses standard sockets for component communication.

2.7 A first example

Let's see in this section an example that illustrates the use of the model mechanisms and shows its adequacy to develop components and applications for ODS. It is the typical application of a farm of processes, where a master has to compute a set of expressions, using worker components to do the job. The idea is to use as many workers as possible from the ones available in our environment.

The scheme of this example constitutes the basis of many distributed applications, that in open systems have to deal with a dynamic number of workers, that can even vary during the application lifetime. The application can be defined in our model in terms of two components, a master and a worker: $A = [\texttt{Master}] \parallel [\texttt{Worker}]^*$. A possible implementation of component `Master` is as follows:

```
public class Master {
  final static int N = ...; // No. of expressions
  Queue expr = new Queue(); // The expressions
  Queue sols = new Queue(); // Their solutions
  ....
  public static void main (String args[]) {
    Init(expr); // Initializes the expressions
    Mailbox mb = new Mailbox(args[0]);
    mb.Send(BCST,"?Compute","0"); // broadcast query
    while (sols.size()<N) { // main loop
      Msg r = mb.Receive();
      if (r.Subj.equals("Re:?Compute")&&(r.Result==OK)) {
        if (expr.size()>0) mb.Send(r.Reply,"!Compute",expr.Get());
      } else if(r.Subj.equals("Re:!Compute")) {
        if (r.Result==OK) sols.Put(r.Info); else expr.Put(r.Info);
        if (expr.size()>0) mb.Send(r.Reply,"!Compute",expr.Get());
      }
```

```
    } // here we have the solutions of the N expressions
  }
}
```

As we can see, component `Master` sends first a message to its domain asking for components implementing method `Compute`, and distributes the tasks as workers answer the messages. On the other hand, component `Worker` deals with the calculation of the expressions, and can be implemented as follows:

```
public class Worker {
  ....
  String Compute(String expr) { .... };
  ....
  public static void main (String args[]) {
    String dest = BCST;
    Mailbox mb = new Mailbox(args[0]);
    if (args.length>=2) dest += "@"+args[1];
    mb.Send(dest,"Re:?Compute","");
    while (true) {
      Msg r = mb.Receive();
      if (r.Subj.equals("?Compute"))
        mb.Send(r.Reply,"Re:?Compute","0",r.Ref);
      if (r.Subj.equals("!Compute"))
        mb.Send(r.Reply,"Re:!Compute",Compute(r.Info),r.Ref);
    }
  }
}
```

Function `Compute()` computes an expression. In both components the first argument is the mailbox name, and component `Worker`'s second argument is the name of the domain where to get tasks from (it can be *living* in a machine but advertizing its services in a bigger domain). It is important to note the first message sent by the `Worker`, that allows the component to dynamically join any application requesting any of the services it offers.

Clear advantages shown by this example are the dynamic configuration of the application, and the independence achieved between the master and the workers. They do not have to statically know each other's identities, neither the master should know the number of workers, that may dynamically vary. And from the worker point of view, it does not care whether there is one or more masters in the domain, it just gets its tasks from whoever requests them.

3 Properties and Controllers

At the beginning of this paper we mentioned some of the specific problems related to ODS. So far we have seen an abstract model of a system, together with a set of basic communication mechanisms between components. However, one of the main contributions of our proposal is based, as previously mentioned, on minimizing the systems and the components requirements, dealing with every specific problem in a modular and independent way through the use of controllers. What happens in the previous example when a `Worker` suddenly stops working? Or, could we reuse the `Master` in an environment where method `Compute` has a different syntax? Or, could we add security controls (e.g. encryption or signatures) to the application?

3.1 Some basic properties

Many behavioral properties of components and applications can be implemented by means of controllers, like security, dynamic reconfiguration, adaptability or some forms of high availability (we refer the reader to the bibliography for a list of properties —or '*ilities*'— that can be implemented using the different reflective approaches cited in there). However, not all applications' requirements can be implemented that way, since reflection on black-box entities presents some limitations when trying to deal with some of the non-functional requirements of the applications, like fault-tolerance, efficiency or timeliness, for instance. Nevertheless, we claim that for those properties that naturally accept a reflective implementation, reusable controllers provide a very general and appropriate solution, with the additional benefits already mentioned: controllers are defined independently from the components they will be later attached to, they are reusable and composible, and easy to build because of their uniform structure.

Among all requirements relevant to the development of applications in ODS, we have selected four examples that can be easily implemented by using controllers. They all try to provide components with some specific feature (*property*) needed for surviving in open and independently extensible environments:

Independence. A component should be self-governed, able to discover the services it needs and free to decide the provider to use. A controller implementing this property maintains a list of the services used by its component, updated with the information from the messages received by the component. When delivering a message, the controller checks whether the destination is active or not, sending always the message to a known active component. The controller is also able to interrogate its environment for valid service

providers. In this sense, the Independence controller is responsible for making the component 'environment-aware'.

Self-Protection. A component should protect itself against external failures and avoid never-ending waits. Controllers of this property use a timeout table for outgoing messages, together with the component instructions for handling timeout conditions.

Adaptability. A component should be extensible and able to accommodate to different interfaces and protocols. Regarding extensibility, controllers of this property find available service providers and re-divert to them the incoming service requests not implemented by the component. Regarding interoperability and interface adaptation, they try to find *translators* for the outgoing messages they handle, components with a similar functionality to Wiederhold's mediators [24].

Integrity. Controllers of this property check pre-requisites on the component interface. Typical examples are user-defined pre-conditions on the incoming and outgoing messages, like control over their partial order or the time intervals between them, or the addition of security controls to the component interfaces, like encryption, signatures, or software licensing mechanisms. Also laws [15] can be defined and enforced using the controllers of this property.

In addition to those, currently implemented in our model, new properties can be defined and implemented using controllers, that can be later reused in other applications (controllers for billing on usage of the components they wrap, for monitoring the access to specific components, service brokers, middlemen, facilitators, etc.).

3.2 Controllers

Controllers are first-class entities that can be attached to mailboxes, capturing their incoming and outgoing messages and modifying them according to their purpose. Each one implements a specific concern or requirement, what we have called a *property*. As we mentioned before, all controllers share the same structure, with their own thread of control and two basic operations, `Received()` and `Deliver()`, that deal with the incoming and outgoing messages, respectively:

```
void Received(Msg m, Queue outq, Queue inq);
void Deliver (Msg m, Queue inq, Queue outq);
```

In them, `m` is the message that has been received or is to be deliver, respectively, and `inq` and `outq` are the mailbox queues of incoming and outgoing messages. Controllers are attached to component mailboxes, and the mailbox is in charge of calling the appropriate methods of the controllers when a message is received or delivered by the component. Each controller will deposit in the mailbox

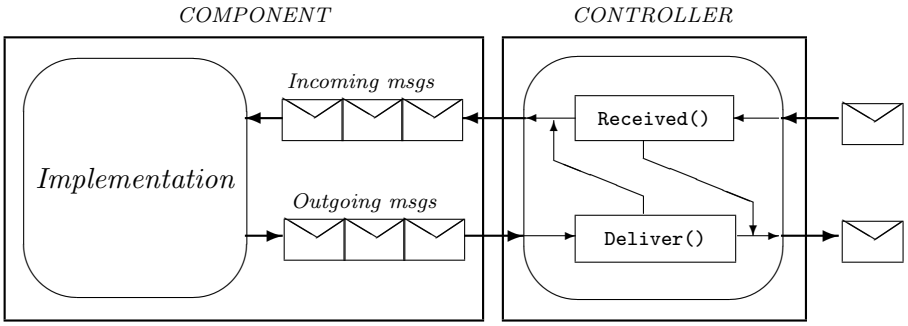

Fig. 2. A component C with its mailbox and a controller.

queues the resulting messages after its treatment (fig. 2).

When multiple controllers are attached to a component mailbox, they are chained in such way that outgoing messages from a controller become incoming messages to its successor. Thus, there is a common way of composing controllers, allowing the enforcement of several properties to a component.

One of our main contributions is to define the controllers to be reusable, so once they are defined and implemented for certain properties, they can be reused when building up applications, adding them to existing components. Users can therefore count on a market of reusable components, together with a market of reusable controllers that implement common properties (adaptability, security, etc). And as well as new components can be developed if no existing component fit our necessities, new controllers can also be developed to fulfill some user specific requirements. Since they all have a common structure, this task gets reduced to specifying the controller basic operations, and our communication infrastructure takes care of the rest of the issues.

3.3 Customizing controllers

In order to be reused, controllers need to be configured to take into account the user specific requirements when attached to a component's mailbox. In the same way controllers share the same structure, they also share the way of being customized. This customization is achieved through the specialization of the class with its *preferences*, a class which contains all the variables and methods configurable by the controller's users. Every controller will define its own specific preferences, although there are two methods common to all controllers, which are defined by the following Java interface:

```
public interface Preferences {
  public boolean relatedMsgIds_In(Msg m);
  public boolean relatedMsgIds_Out(Msg m);
}
```

These methods determine the (incoming and outgoing) messages that a controller deals with, letting the rest of the messages pass through it unmodified.

3.4 Implementing controllers

Controllers can be implemented by simply providing the body of methods `Deliver` and `Received`, that determine their behavior, and by defining the class with their preferences. The following Java class corresponds to the simplest case of a controller that let all messages pass unmodified. All controllers inherit from this class.

```
public class Controller {
  private static NumControllers = 0;
  int Cid;
  Preferences Prefs;
  public Controller () {Cid = ++NumControllers;}
  public void setPreferences (Preferences P) {Prefs = P;}
  public void Received(Msg m, Queue outq, Queue inq) {inq.Put(m);}
  public void Deliver (Msg m, Queue inq, Queue outq) {outq.Put(m);}
}
```

To show an example of the implementation of controllers, let us describe here with more detail one of them. For space limitations we have chosen the controller of the property of Self-Protection. Although a quite simple and straightforward controller, it allows to show concisely how controllers can be implemented. As previously mentioned, the controller of this property maintains a list of the component's outgoing messages, which is updated every time a message is sent out or a reply is received for one of them. A clock is also started for every outgoing message with a timeout condition. If the timeout is reached without having obtained a response for that message, the controller will produce a reply to the component, indicating that situation. The basic structure of this class is as follows:

```
public class SelfProtectionController
      extends Controller implements Runnable {
  private Queue Pending, SLEEP, INQ;
  public SelfProtectionController() { ...class constructor... }
  public void Deliver(Msg m, Queue inq, Queue outq) { .... }
  public void Received(Msg m, Queue outq, Queue inq) { .... }
  public void run() { .... }
}
```

As we can see, this class inherits from general class `Controller`, and it also has its own thread of control, therefore implementing interface `Runnable` and defining method `run()`. Its attributes are the list of outgoing messages waiting

for a response (`Pending`), a message queue for synchronizing the controller's thread with the clocks started for every message (`SLEEP`), and the component's incoming message queue where the controller should deposit the reply it builds in case a timeout is reached (`INQ`).

Coming down to the methods of this class, the first one is the constructor, that simply initializes the attributes and starts the controller's thread:

```
public SelfProtectionController() {
  super(); Pending = new Queue(); SLEEP = new Queue();
  Thread t = new Thread(this); t.setDaemon(true); t.start();
}
```

Methods `Received` and `Deliver` are the ones that deal with the incoming and outgoing messages of the controller, respectively. They use the user's preferences `Prefs` for determining the messages to be filtered, assigning timeouts, and recognizing their responses:

```
public void Received(Msg m, Queue outq, Queue inq) {
  if (Prefs.relatedMsgIds_In(m)) {
    int i = 0;
    while (i < Pending.size()) {
      Msg p = Pending.Element(i);
      if (Prefs.IsTheReply(p,m)) Pending.Delete(i);
      else i++;
    }
  }
  inq.Put(m);
}

public void Deliver(Msg m, Queue inq, Queue outq) {
  INQ = inq; int tout;
  if (Prefs.relatedMsgIds_Out(m)) {
    if (m.ReplyBy>m.Sent)
      tout=(int)(m.ReplyBy-m.Sent);
    else tout = Prefs.setTimeout(m);
    if (tout>0) { // starts a clock
      m.ReplyBy = m.Sent+tout;
      new Sleeper(tout,SLEEP,m);
      Pending.Put(m);
    }
  }
  outq.Put(m);
}
```

Method `Deliver` uses another class to start the clocks. Each object of this `Sleeper` class simply sleeps the specified period, and then deposits the given

message into the SLEEP queue. This queue is used by the controller's thread of control to know about the end of the clocks:

```
public void run() {
  while (true) {
    Msg m = SLEEP.Get(); // blocked until a timeout occurs
    int i = Pending.Find(m);
    if (i!=-1) {Pending.Delete(i); INQ.Put(Prefs.answerTout(m)); }
  }
}
```

In order to finish with the description of the implementation of the controller of SelfProtection, we need only to define the class with its preferences:

```
public class SelfProtectionPreferences implements Preferences {
  public boolean relatedMsgIds_In(Msg m)
    {return false;} // no incoming msgs related
  public boolean relatedMsgIds_Out(Msg m)
    {return false;} // no outgoing msgs related
  public boolean isTheReply(Msg r,Msg m) //does r reply to m?
    {return r.RefTo == m.Ref;}
  public int setTimeout(Msg m) // timeout assigned to msg m
    {return 0;}
  public Msg answerTout(Msg m) { // builds the reply to msg m
    Msg r = new Msg(); r.Subj = "Re:"+m.Subj; r.Result = TIMEOUT;
    r.RefTo = m.Ref; r.Info = m.Info; return r;
  }
}
```

Users will only have to override those methods for configuring the controller's behavior according to their preferences.

3.5 A real application

To show an example of the use and the capabilities of controllers, let us consider a typical coarse grain massive parallel computation with 10^4 to 10^8 large records that have to be independently analyzed, each one requiring considerable computational effort. One common idea to solve this kind of problems is based on using the idle times of LAN, WAN (or Web!) interconnected workstations, trying to make use of their potential CPU power, currently wasted. In [10], K. Jeong gives a good introduction to the problem, many references to different solutions, and his own solution based on a fault-tolerant approach, using Linda, checkpointing and transactions. What we shall see here is how the problem can be solved using our model.

In the first place, the solution to the problem can be expressed as a master-worker application, where a master holds the data to be evaluated, and multiple workers do the computations, following the scheme described in section 2.7. The question is how to reuse the components defined in that section, and how to incorporate the application specific requirements to them.

We know that worker processes will run on different workstations (or PCs or Macs) and that the master generates tasks for them. Each worker is created when the WS detects that is idle, and abruptly eliminated as soon as the WS is busy again. The reason for not letting the worker finish its current task is to ensure that a user has his WS available as soon as he requests it; in that way, detection of activity in a WS is equivalent to its sudden (unrecoverable) failure from the master's point of view: in any case there is no answer from the worker. This is why this problem has been usually treated from the fault-tolerance point of view.

But in our case it is enough to provide the master component with the property of Self-Protection, that deals with the absence of replies. In this way, any outgoing message requesting a computation would be assigned a timeout. If the timeout expires, the master can conclude that the worker has been eliminated and it replies back to the master with an error, forcing the corresponding task to be put back into the master's pending list. The way of configuring this property to our particular requirements is by simply specializing the class with the Controller preferences:

```
public class MySPPrefs extends SelfProtectionPreferences {
  public boolean relatedMsgIds_Out(Msg m)
      {return m.Subj.equals("!Compute");}
  public boolean relatedMsgIds_In(Msg m)
      {return m.Subj.equals("Re:!Compute");}
  public int setTimeout(Msg m) {return 10000;}
}
```

To incorporate this property to the master, it is enough to attach the controller to the component mailbox, replacing the previous way of creating the component mailbox with:

```
  Vector Controllers = new Vector();
  SelfProtectionController SP = new SelfProtectionController();
  SP.setPreferences(new MySPPrefs());
  Controllers.addElement(SP);
  Mailbox mb = new Mailbox(args[0],Controllers);
```

As we can see, mailboxes admit a second argument during their creation: a vector with the controllers that we want to attach to them.

There is also another case to cover in the application: when a communication delay makes the master think that the worker is dead while the reply is on its way. To cope with this potential problem it is enough to provide the master with the property of Integrity, that does not allow duplicated responses. Other interesting use of this property is to add some sort of security controls, e.g. message encryption.

Summarizing, we are able to solve the problem by reusing components, and treating each requirement in a modular way. Moreover, the properties used here are already developed, so we do not need to develop any specific software for the application.

Comparing our solution with the ones proposed by other authors, we see that what really complicated other solutions was the intrinsic nature of the problem, typical of ODS. Since our model is specifically designed to cope with these issues, the benefits we obtain are important. In the first place, there is no need for the master to register or monitor the workers, whose number can be absolutely variable. Second, the application's fault tolerance can be greatly simplified by just discarding late replies, independently from the type of failure the workers may experience. And finally, the sort of application (coarse grain and parallel) suits very well our communication mechanisms and the Internet open and distributed infrastructure, as opposed to other approaches based on shared memory.

What we do not get with this solution is partial recovery of the workers' job. Once a worker is eliminated all its work is lost, since we have not contemplated checkpointing and state recovery in the components of the example. However, the problem can be solved by the use of passive replication if the components can incorporate these two mechanisms, through the use of the property of High Availability. On the other hand, multiple masters can co-exist in a domain, dynamically sharing the workers.

4 Building applications

Using **SC**, applications are built mainly from existing components, that are customized through the use of controllers to incorporate the context-specific requirements. Only those components and controllers not available need to be developed, always encapsulating just the computations into the components, and leaving the rest of the concerns to the controllers. This is a well-known methodology for building applications, currently defended by most software architects. However, once the components and the controllers of our application are identified (or at least, their initiators), we need to provide some sort of tool support for their deployment, configuration and execution in a particular

domain.

4.1 CDL: A description language for components and controllers

Once all the parts of the application are defined, we need some sort of description language for identifying, configuring and composing them. The main purpose of the description language is to provide the users of components and controllers with a high-level definition that abstracts the relevant aspects of their interfaces and configuration parameters. We have therefore defined CDL, a language for the description of the components and the controllers interfaces. It is devised to be used by both application developers and the run-time system. Application developers use it for identifying the services offered by a component, learning about the way a controller modifies the interface of a component, and expressing the composition of controllers and their addition to a given component. On the other hand, the run-time system uses CDL to configure and build the final executable components out of the basic components and controllers. It is also used among components for describing their interfaces, when asked for them by special message selector "??".

CDL will be introduced by examples, using the components and controllers defined in the previous application. For instance, the CDL description of component `Master` is as follows:

```
Component Master {
  CoreComponent: Master;
  Controllers: {};
  Parameters: String Mailboxname;
  Out: ?Compute();
       !Compute(String expr);
  In: Re:?Compute();
       Re:!Compute(String sol);
}
```

Analogously, the description of component `Worker` is the following:

```
Component Worker {
  CoreComponent: Worker;
  Controllers: {};
  Parameters: String MailboxName;
              String DomainName = "";
  Out: Re:?Compute();
       Re:!Compute(String sol);
  In: ?Compute();
       !Compute(String expr);
}
```

The names of the components are at the top, followed by the 5 parts of their descriptions. The first one (`CoreComponent`) identifies the name of the original component that was wrapped with the controllers listed in part `Controllers` to produce the current component. Part 3 (`Parameters`) describes the arguments of the component as a set of variables and/or functions (e.g. its mailbox name), that must be given when executing the component. The last two parts describe the component interface in terms of their incoming and outgoing messages. At this level only the message selectors and their arguments are relevant.

On the other hand, the controller of Self-Protection is described as follows:

```
Controller SelfProtection {
  Preferences:
    boolean relatedMsgIds_In(Msg m);
    boolean relatedMsgIds_Out(Msg m);
    boolean isTheReply(Msg orig,Msg answr);
    int setTimeout(Msg m);
  Received:
    !f -> {!f}c, {-}e;          Re:!f -> {Re:!f}c, {-}e;
    ?f -> {?f}c, {-}e;          Re:?f -> {Re:?f}c, {-}e;
  Deliver:
    !f -> {!f}e, {Re:!f,-}c;    Re:!f -> {Re:!f}e, {-}c;
    ?f -> {?f}e, {-}c;          Re:?f -> {Re:?f}e, {-}c;
}
```

The name of the controller is at the top, followed by 3 parts. The first one contains the set of functions that determine the user preferences, while the last two parts of the controller description determine its interface. Part `Received` deals with the messages received from the environment. For each possible message selector received (`!f`, `?f`, `Re:!f`, and `Re:?f`) the controller produces two lists of message selectors: those which are handled to the component (`{}c`), and those returned to environment by the controller itself (`{}e`). In this case, all received messages are passed to the component unmodified. Part `Deliver` deals with the messages that the component wants to send out. For each message selector it also produces two lists, with the same meaning. Items in the lists indicate the different alternatives the controller may produce, and "`-`" means 'no output'.

4.2 Wrapping

The interface parts of this description may also serve as a *set of rules* that describe the changes in the component behavior when wrapped by a controller. Therefore, given the descriptions of a component C and a controller

L, the description of the new component D obtained by wrapping C with L is obtained as follows:

(1) The name of the new component is D, the component stated in part `CoreComponent` is C, and L is added at the end of the list in part `Controllers`.
(2) D parameters are the same as C's.
(3) Message selectors in sections `In` and `Out` are obtained by applying the changes described in L sections `Received` and `Deliver` to C message selectors.

In the example, the component obtained by wrapping the `Master` with the controller of Self-Protection can be described in CDL as follows:

```
Component SPMaster {
  CoreComponent: Master;
  Controllers: SelfProtection[MySPPrefs];
  Parameters: String Mailboxname;
  Out: ?Compute(); !Compute(String expr);
  In: Re:?Compute(); Re:!Compute(String sol);
}
```

The preferences class is indicated in square brackets after the corresponding controller name.

4.3 Putting all pieces together

Once we have the CDL description of all components and controllers, the runtime system is able to build the application from them. Each component is executed with the "`launch`" command in the chosen machine, specifying the name of the component and the actual value of its parameters.

For instance, the command "`launch SPMaster mbox`" executes the `Master` component with the property of Self-Protection. This command examines first the file `SPMaster.cdl`, with the CDL description of this component, and pulls along the appropriate java classes (components and controllers), building the final component (`SPMaster.class`) from them. This class is then executed on the Java virtual machine with the given parameters. The rest of the workers can be launched in a similar way in the machines we want them to execute, if not already running. Please note that this is an alternative way of adding controllers to (binary) components, instead of attaching them to the mailbox in the actual component's code when creating it, as shown in section 3.5.

On top of those basic facilities, we are currently developing a visual IDE (Integrated Development Environment) that includes palettes for displaying

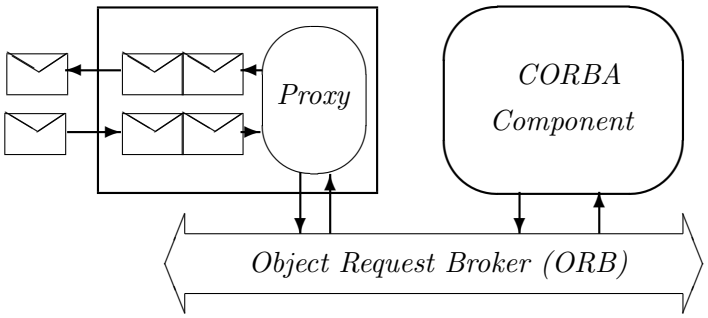

Fig. 3. A Proxy for a CORBA component.

as icons the components available in a given domain; a 'canvas' container onto which components are configured and placed to build up applications; a browser for inspecting and locating components that match some user defined search criteria; and access to an object-oriented editor and some compilers for developing new components and include them into the IDE.

5 Other Considerations

Once we have the overall picture of the model, let us discuss some other issues of special interest to the model and to its potential users.

5.1 Interoperability

Having (yet) another component model is not useful unless we offer smooth integration with the rest of the component models. Bridges need to be defined to other technologies and to existing legacy systems in order for them to seamlessly interoperate.

Our model was originally designed to express and implement, in a natural way, most of the specific requirements of ODS applications. However, we also decided to keep the model as simple, neutral, and general as possible. The reason behind this decision was that simple and neutral models are easy to implement, to reason about them formally, and to integrate with other component models.

For interoperation we have chosen proxies that serve as gateways to the components of other models and systems. Therefore, if we want to use a CORBA component, for instance, and incorporate it into an application built using our model, we just need to create a proxy for it (fig. 3). A proxy of a CORBA component will be a CORBA component but with an **SC** mailbox, thus being able to talk to both worlds. In this way components from other models

or platforms can also benefit from the specific features of our model, such as inspection, local broadcasts, and the use of reusable controllers for customizing their behavior. Properties (such as Independence, Self-Protection, etc.) can be also added to third-party *alien* components by simply attaching the appropriate controllers to their proxies.

The use of proxies provides a smooth form of interoperation between different kinds of components. In the particular case of our master-worker application, we have defined CORBA, JavaBeans and COM components implementing the method `Compute()` and made them interoperate easily with **SC** workers to get the master's tasks done.

5.2 Formal support and methodology

There are many advantages derived from defining a simple and neutral component model like **SC**. In this section we will cite two among them, although we will not discuss them in detail for being outside the scope of this paper.

In the first place we have the possibility of using formal methods for reasoning about the components, the controllers, and the applications built using the model. In this sense, **SC** counts with a formal framework that uses Object-Z [8] and temporal logic for specifying and supporting the model and its mechanisms [20]. For instance, our theory provides the developer of a controller with formal support to determine the effects of wrapping a component with it, compose it with other controllers, and formally characterize the applications for which the wrapped component can substitute the original one.

Another benefit of our model comes from the fact that controllers share a uniform structure. Based on that, it is possible to define a methodology for their systematic construction. That methodology is the subject of ongoing research, and covers all their life-cycle, defining processes for guiding the specifications of the controllers in the formal framework, reasoning about their behavior, deriving their description in CDL, and implementing them.

5.3 Limitations

In general, the reflective approach followed by our model with independent layers (the controllers) for implementing the properties has some known limitations. For instance, requirements that affect the internal behavior of components (like efficiency), *hard* requirements (such as fault-tolerance or timeliness), or those that cross-cut different dimensions of the applications, are difficult to manage with an *onion*-like layered approach, as pointed out by

Aspect-Oriented Programming [13]. Nevertheless, it is a very good approach for building applications from reusable back-box components, where users are not willing or cannot change their internals. Besides, most ODS requirements can be implemented with a combination of controllers and specialized components, the only entities in our model (we requested systems to offer *just* the infrastructure for the creation and communication of components). In this sense, transactions and algorithms can be encapsulated into specialized components, which is, by the way, a good design practice for ODS applications since it helps producing more modular and maintainable applications. And also *soft* versions of some of the hard requirements can be easily achieved (using properties like Self-Protection or High Availability). The characterization of the properties that can be implemented using a layered reflective approach like this is still an area of ongoing research.

On the other hand, we have initially restricted our model to a syntactic level of interoperation between components (i.e. signature of operations), the same level the rest of the distributed component platforms currently support. However, interoperability at the *protocol* (partial ordering between exchanged messages and blocking conditions) and *semantic* (meaning of operations) levels is now recognized as necessary for building up applications in ODS. The provision of a semantic framework for components to interoperate at these levels is yet to be explored.

6 Related Work

The idea that originated this work was born from the original papers from Tokoro and Takashio [19], and from Minsky and Leichter [15]. The former mention real time, asynchrony and autonomy as key issues in ODS, but only deal with the first two in their paper, without paying attention to autonomy. On the other hand, Minsky's Law-Governed Architectures can be seen as a one-property scheme defined for a shared memory model (Linda), that we have extended and generalized to produce our general model.

Layered reflective models are well known and widely used. Apart from the two aforementioned papers, we can also cite the Composition Filters model [3], the Layered Object model LayOM [5], the meta-actors defined in the Actors model [1], the object filters [11], the MetaCombiner approach by Mezino [14], COM's containment [17], Orbix filters [4] or the message handlers in Oberon-2 [16], among others. However, our controllers are first-class reusable entities, they all have the same structure, and are more than just computational filters, while the meta-components of those models suffer from the limitations we pointed out in the introduction of this paper.

Regarding the properties, there exists a lot of literature about them in the field of Artificial Intelligence, where heated discussions take place about the precise definition of Autonomous Agents and their properties [9,12]. However, apart from informal definitions we do not know of any other work that try to formally define and specify them. The properties we have mentioned here are well known, and many systems and components already implement them, although usually hard-wired into the components' or the systems' code. The model we have introduced here has proven its powerfulness by being able to specify them in a natural way, and implement them in a unified and modular fashion.

Neither the layered model or the communication mechanisms used in **SC** try to be a novelty, but to be general, simple, and expressive enough for our purposes. Other approaches use different communication paradigms, such as shared-memory or ports and channels. The PageSpace coordination model [7] and the Infospheres project [6] represent both alternatives. The first one combines Linda tuple spaces, services, and Web technology to build up a coordination model for Internet applications using a shared-memory approach. The Infospheres 1 project uses message channels and ports to compose components, and has been enhanced to Infospheres 2, that uses temporal logic to reason about interactions of dynamically reconfigurable components communicating through RPC, messages and event-oriented middleware. However, our approach is based on distributed components that use message passing to communicate among themselves. Services are not located in a shared places but *owned* by other components, and mailboxes are used instead of channels. We think that mailboxes are more appropriate to deal with the ODS dynamic changes, and the main drawback of this paradigm, the need of being aware of the targets identities, can be easily and naturally solved in our model using the appropriate controllers (e.g. Independence). In this respect our work is similar to the Actors model [1], although our reusable controllers try to overcome some of the limitations that meta-actors currently have: dependability on the actors they wrap, lack of common structure, and difficult reusability.

Finally, our model offers a much more lightweight compositional infrastructure when compared to existing middleware component platforms, such as CORBA, DCOM or Java Beans. The main aim of our model is to serve as a neutral platform with the minimum set of features required for developing distributed applications in open systems, but expressive enough to do so in a natural way. We do not try to compete with them in those areas and specific tasks where they are more powerful than any other model, but to integrate and complement them where they have shown their weaknesses. This is why interoperability with other models is so important in **SC**.

7 Conclusions

Component-Oriented Programming (COP) has been described as the natural extension of Object-Oriented programming to the realm of independently extensible systems. COP aims at producing software components for a global component market and for late composition [22]. However, the design of those components is challenging the software community with its specific issues. Reusability and late composition are two driving forces towards the separation of the computational and interoperational aspects of components, while ODS-specific requirements have to be incorporated into the user application too, in an orderly manner. Our contribution tries to address these goals, offering a component model that allows the modular, reusable and extensible composition of components in open and distributed environments to build up applications. It is based on the separation of the computational aspects of components from their interoperational and context-specific concerns, encapsulating the latter into reusable entities (called controllers), an extension of traditional object wrappers that try to overcome many of their limitations.

Our work is motivated by a more ambitious project that aims for the development of reliable e-commerce and multimedia applications for open systems, with the specific intention of providing the electronic access to our Faculty resources, systems and premises. **SC** tries to serve as the basic model where both existing and new components can be integrated to build up the required applications.

With regard to the example used in this paper for showing the expressiveness and adequacy of our model, it constitutes the skeleton of many real and useful applications. In [10] an elementary particle physics problem is cited, while it can also be used in other NP-hard problems, like first-order logic theorem proving, whose natural complexity can be approached by using distributed solutions based on the master-worker approach. With our model we also obtain a simple solution that can easily take advantage of the idle times of Internet-connected workstations. The expressiveness of the model has also been tested with other typical distributed applications, like meeting schedulers or distributed auction bidding, that can be easily written using the model mechanisms.

Ongoing research on the model is focused on three main areas. In the first place, we want to study the interoperability of components at the protocol and semantic levels, whereby components could find the services they need by their specifications, and not just by their names. Second, we want to characterize the ODS specific requirements and '*ilities*' that can be implemented using our reusable wrappers, and the ones that cannot. And finally, we plan to finish soon the methodology for the systematic construction of the reusable controllers.

Besides, we continue improving the current prototype, building a more efficient implementation based on the users' feedback and new requirements.

References

[1] G. Agha, Abstracting interaction patterns: A programming paradigm for open distributed systems, in *Proc. of FMOODS'97* (Chapman & Hall, 1997).

[2] G. Agha, Compositional development from reusable components requires connectors for managing both protocols and resources, in *Proc. of the Workshop on Compositional Software Architectures* (California, January 1998).

[3] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, Abstracting object interactions using composition filters, in *Proc. of ECOOP'93*, number 791 in LNCS (Springer-Verlag, 1993) 152–184.

[4] S. Baker, *CORBA Distributed Objects*, (Addison-Wesley Longman, 1997).

[5] J. Bosch, Language support for component communication in LayOM, in: M. Mühlhäuser, ed., *Workshop Reader of ECOOP'96* (Dpunkt Verlag, 1996) 131–138.

[6] K. M. Chandy, A. Rifkin, and The Infospheres Group, Caltech infospheres project, in *Proc. of the Workshop on Compositional Software Architectures* (California, January 1998).

[7] P. Ciancarini et al., Coordinating multiagent applications on the WWW: A reference architecture, *IEEE Trans. on Software Engineering*, **24** (May 1998) 362–375.

[8] R. Duke, G. Rose, and G. Smith, Object-Z: A specification language advocated for the description of standards, Technical Report number 94–45 (University of Queensland, December 1994).

[9] S. Franklin and A. Graesser, It is an agent, or just a program?: A taxonomy for autonomous agents, In *Proc. of the 3rd International Workshop on Agent Theories, Architectures and Languages*, number 1193 in LNCS (Springer-Verlag, 1996).

[10] K. Jeong, *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing and Transactions*, PhD thesis (Dept. of Computer Science, New York University, 1996).

[11] R. Joshi, N. Vivekananda, and D. J. Ram, Message filters for object-oriented systems, *Software–Practice and Experience*, **17** (June 1997) 677–699.

[12] D. Kafura and J. Briot, Actors and agents, *IEEE Concurrency*, (April-June 1998) 24–29.

[13] G. Kiczales et al., Aspect-oriented programming, in *Proc. of ECOOP'97*, number 1241 in LNCS (Springer-Verlag, 1997) 220–242.

[14] M. Mezini, Dynamic object evolution without name collisions, in *Proc. of ECOOP'97*, number 1241 in LNCS (Springer-Verlag, 1997) 190–219.

[15] N. Minsky and J. Leichter, Law-governed Linda as a coordination model, in: P. Ciancarini, O. Nierstrasz, and A. Yonezawa, eds., *Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems*, number 924 in LNCS (Springer-Verlag, 1995) 125–146.

[16] H. Mössenböck, *Object-Oriented Programming in Oberon-2* (Springer, 1995).

[17] R. Sessions, *COM and DCOM. Microsoft's Vision for Distributed Objects* (John Wiley and Sons, 1998).

[18] C. Szyperski, *Component Software* (Addison-Wesley, 1998).

[19] M. Tokoro and K. Takashio, Towards languages and formal systems for distributed computing, in *Proc. of ECOOP'93*, number 791 in LNCS (Springer-Verlag, 1993) 93–110.

[20] J. M. Troya and A. Vallecillo, Specifying reusable controllers for software components, in *Proc. of FMOODS'99* (Kluwer Academic Publishers, February 1999) 131–148.

[21] S. Vinoski, New features for CORBA 3.0, *Commun. ACM* **41** (October 1998) 44–52.

[22] W. Weck, J. Bosch, and C. Szyperski, Summary of WCOP'97, in *Proc. of the ECOOP'97 Workshop on Component Oriented Programming (WCOP'97)*, number 1357 in LNCS (Springer-Verlag, 1997).

[23] I. Welch and R. Stroud, Using metaobject protocols to adapt third-party components, in *Proc. of Middleware'98* (March 1998).

[24] G. Wiederhold, Mediation in information systems, *ACM Comp. Surveys*, **27** (June 1995) 265–267.