An Infrastructure for Adaptable Middleware*

Pierre-Charles David and Thomas Ledoux

École des Mines de Nantes
Département Informatique
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
{pcdavid,ledoux}@emn.fr

Abstract. Today's software systems have to deal with an increasing diversity and complexity of execution environments. Next generation applications will have to deal with the unknown, with execution conditions which can not be predicted at the time they are written: they must be *adaptable*. In this paper, we present our current answer to this problem, in the form of an infrastructure for adaptable middleware. This infrastructure uses a Separation of Concerns approach where the associations between functional and non-functional components can be modified at run-time. These associations are controlled by an *adaptation engine* which monitors both the execution environment and the application, and adapts the associations according to *adaptation policies*.

1 Introduction

Today's software systems have to deal with an increasing diversity and complexity of execution environments. New types of computing platforms appear regularly like Personal Digital Assistants and mobile phones, each with different processors, Operating Systems, hardware devices, etc. This context makes applications development more and more difficult: programmers need to deal with a wide spectrum of hardware platforms, each with dynamically varying – and generally unpredictable and uncontrollable – resources, and which all have to work together. Next generation applications will have to deal with the unknown, with execution conditions which can not be predicted at the time they are written: they must be able to adapt their behavior to fit the dynamically evolving environment.

Adding ad-hoc support for dynamic reconfiguration and adaptation in an application is not a trivial task. It can result in very complex systems where the real business features are hidden by all the mechanisms introduced to allow appropriate dynamic reconfigurations. In our opinion, the most promising approach to solve this problem is to build an insulation layer, the *middleware*, between the applications and their environment. This layer provides common abstractions and services for applications developers to write their programs, and, most importantly for our problem, factors all the environment-specific code

* This research is supported by the RNTL project ARCAD (http://arcad.essi.fr)

outside from the main – functional – code. This is a first step towards a solution for dynamic adaptation, because changes in the execution environment now mostly impact the middleware layer instead of the whole application. The next step is to make this middleware more flexible, and able to be adapted – or even better to adapt itself – to environmental changes. Although current industrial middleware implementations ([1, 2, 3]) do not take into account the dynamic execution conditions, previous works on *adaptable middleware* [4, 5] have shown the viability of this approach.

In this context, the goal of our work is to make it easy for the application developers to build adaptable applications, and to provide the required infrastructure and tools to actually adapt them. The solution we propose in this paper uses a Separation of Concerns approach to make explicit the separation between what depends on the changing environment (non-functional, middleware services) and what does not (functional, business components). Then, the associations between these two kinds of code can be modified at run-time in a way that depends on dynamic execution conditions, to adapt the application to this environment.

This paper is organized as follow: first we present a simple, concrete example of an adaptation scenario that will be used an an illustration thoughout the paper (Sect. 2). Then, we describe the general architecture of our infrastructure (Sect. 3) before detailing each of its subsystems (Sections 4 to 6) and showing how the current prototype is used in practice (Sect. 7). Finally, we compare our approach to other related works (Sect. 8), before concluding (Sect. 9).

2 Running Example

This section describes an example application which will be used througout the paper to illustrate our proposition.

The application is a client/server based bookstore system. A bookstore company provides its customers a Java-based graphical client to allow them to browse and search the catalog, to order books and to manage their accounts. The company expects its customers to use the clients in all kinds of situations and on different platforms (desktop computers, laptops, even PDAs), and wants their experience to be as smooth as possible. On the server side, the programmers would like to use server resources efficiently, but they don't want to have to deal with too many low-level details (from their point of view), like persistence of their objects for example.

The business model contains classes such as `Catalog`, `Book`, `Order` and `Customer`. Using this core classes, the company programmers create two applications. The first one is a server designed to be used internally to manage the catalog and follow customers' orders. The second one is the graphical client that is shipped to customers. None of these two programs contain any reference to middleware-type services like server-side persistence or distribution mechanisms; these will be added separately to enable their dynamic adaptation, as will be shown in the rest of the paper.

3 Proposed Architecture

In this section, we give a brief overview of the architecture we propose to enable application adaptation using a middleware-level infrastructure. More details on each of the subsystems introduced here can be found in the next sections (Sect. 4, Sect. 5 and Sect. 6).

From a very abstract perspective, we can consider any software system as being the implementation of a *solution* to a given *problem* (specification documents and functional requirements) in a particular *context* (available resources, non-functional requirements). This means that our system depends both on the problem specification and on the execution context. The problem definition tends to stabilize over the time, even though it is never completely freezed. On the other hand, the execution context is constantly changing, and at different rates: from macroscopic evolutions (new hardware platforms with specific characteristics appearing every few months), to microscopic changes (resources availability with evolutions in the order of a second or less).

The biggest challenge is thus to design and build software systems so that they can be adapted to dynamically varying resources, and this is why we have chosen to concentrate our work on this kind of adaptations. As we said in the introduction, we think that middleware approaches, with the explicit separation of *functional code* (resulting from the problem specification) and *non-functional code* (low level services dealing with the execution context) are a natural target for adaptations to the execution conditions.

The architecture we propose (see Fig. 1) roughly consists in three parts, corresponding to its three main functions: observation (of the environment and of the application), decision taking, and action (on the running system). A simple *monitoring framework* (Sect. 5) is used to detect changes in the execution environment, and introspection is used to observe the application. When a significant change is detected, the *adaptation engine* is notified. This component is in charge of deciding what modification is necessary to adapt the system, based on *adaptation policies* written in a custom language (Sect. 6). Finally, the decision taken by the adaptation engine is implemented on the running system by modifications of the associations between functional and non-functional components (Sect. 4).

The following sections present theses three subsystems in more details.

4 Application Model and Available Actions

This section describes the application model from the user's point of view, and the possible reconfigurations enabled by this model.

We consider that the application is composed of two kinds of components: functional ones (derived directly from the problem specification, in our example `Customer`s and `Book`s for instance), and non-functional ones (providing execution mechanisms and services adapted to a particular environment, for example distribution using RMI or SOAP). The modifications our system can perform

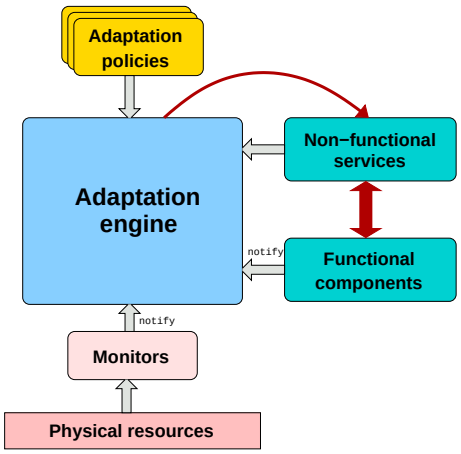

Fig. 1. Global architecture

concern the run-time associations between functional and non-functional components.

4.1 Functional Components

Because we didn't want to impose any unnecessary constraints to the application programmer, what we call functional components here are actually simple, completely standard Java objects. At least, from the user's point of view. When he develops his business code, the application programmer does not need to do anything special like implementing required interfaces, or to conform to any standard or conventions (like EJB's requirements for explicit interfaces specification).

If we take the example presented in Section 2, the `Catalog`, `Book`, `Customer` and `Order` classes would be completely standard Java, without any reference to external libraries.

However, in order to be able to adapt the behavior of these objects, our infrastructure needs to get some control over their execution. To get this control, we use reflective techniques: every functional component gets associated with a meta-level controller which intercepts and interprets all objects creation, method receptions, fields accesses. The meta-level infrastructure is based on the RAM Meta-Object Protocol [6]. The meta-level controller associated to each functional component is an instance of class `Container`, subclass of RAM's default `MetaObject` class (see Figure 2 to get an idea of this class's interface). The `Container` class, named so because of its similarities in role with Enterprise Java Beans containers [1], adds to this default meta-object class the support required for dynamic composition of services and is described in section 4.2.

In order to introduce the indirection to the meta-level into the standard Java code written by application programmers, we provide a special compiler de-

```
public class MetaObject implements Serializable {
    public Object create(ReflectiveObject creationRequestor,
                         Constructor constructor, Object[] initArgs);
    public void initialize(ReflectiveObject base, Object[] parameters);
    public Object invokeMethod(ReflectiveObject receiver, Method method, Object[] args);
    public Object getField(ReflectiveObject target, String fieldName);
    public Object setField(ReflectiveObject target, String fieldName,
                           Object oldValue, Object newValue);
    public void deserialize(ReflectiveObject obj, ObjectInputStream in);
    public void serialize(ReflectiveObject obj, ObjectOutputStream out);
}
```

Fig. 2. Outline of the RAM MetaObject protocol

scribed in [7] (actually a source preprocessor using AspectJ [8]) which transforms classes written by the user into RAM `ReflectiveObject`s. The transformation consists in the addition of a new field (the link to the meta-level `Container`), and the redirection of every method call, object creation and field access to this metaobject (the original methods are renamed and made transparently accessible from the meta-level).

To summarize, from the point of view of the application programmer functional components are standard Java objects, except that they must be compiled using a special compiler. Behind the scene, this compilation adds reflective features to these objects using the RAM Meta-Object Protocol, so that their behavior can be observed and controlled at run-time.

4.2 Non-functional Components: Services & Roles

The second part of our application model deals with non-functional components. These are meant to implement middleware-type services, like distribution and persistency, and to be applied upon functional components.

Figure 3 shows the classes used to model non-functional components in our system.

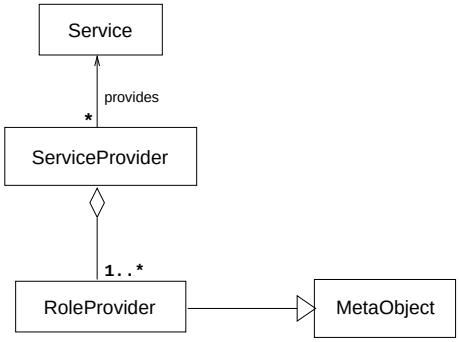

Fig. 3. Services and roles

`Service` Represents an abstract non-functional service, identified by a name. Objects of this class currently serve only as identifiers and do not contain any implementation. This allows for multiple implementations of the same service, each of which can have different properties. *Examples*: `replication.passive`, `distribution`.

`ServiceProvider` Represents a specific implementation of a service. For example, a distribution service for our bookstore example application, implemented using RMI, would include an `RMIProvider` class specializing `ServiceProvider` which would declare implementing the `distribution` service.

`RoleProvider` Meta-objects providing the actual service implementation.

In the current prototype, we consider only adaptations of the behavior of functional components that can be done by modifying how the messages it receives are interpreted. These adaptations can be implemented by specific meta-objects, the `RoleProvider`s. Although for very simple services like tracing only one component is concerned, for most services a cooperation between multiple components is necessary. For example, in a remote message sending service, at least two components must cooperate: a server and a proxy. Our system sees a service as a set of cooperating roles. Each of these roles is implemented by a specific `RoleProvider` meta-object, and these are coordinated by the corresponding `ServiceProvider`. In the previous example, the `RMIProvider` service defines two roles, named `server` and `proxy`. The usage of this service implies the use of both roles.

The `Container` attached to every functional component is a "generic" meta-object in the sense that it does not in itself modify the default interpretation of the messages it intercepts. However, it provides the ability to compose multiple `RoleProvider`s (which are meta-objects) and to modify this composition dynamically[1]. To do this, `Container`s manage a dynamic list of `RoleProvider`s attached to them, and provide a simple protocol to allow the dynamic manipulation of this list (see Fig. 4). This allows a fine-grained adaptation, because every single functional component can be adapted independently, as opposed to system-wide adaptations found in [5] for example, or even class-wide adaptations.

When a `Container` intercepts a message sent to its underlying component, it reroutes the message to all the roles currently attached. Each non-functional component can then process the message in order to implement the non-functional service it corresponds to.

The result of this system is that the behavior of each functional component can be modified at runtime by attaching or detaching the appropriate `RoleProvider` to its `Container`.

[1] Currently, we do not handle the problem of the roles composition; they are simply chained and each one is called in turn.

```
public interface Container {
  ...
  RoleProvider getRoleProvider(Service service, String role);
  void attach(RoleProvider rp);
  RoleProvider detach(Service service, String role);
  ...
}
```

Fig. 4. Roles manipulation protocol of the `Container` interface

5 Observation

Adaptation must be guided by a good knowledge both of the execution environment and of the system itself. The observation framework described in this section has two roles: to expose enough information to the adaptation engine to allow it to take educated decisions, and to detect meaningful changes in these informations. Of course, which changes are meaningful depends heavily on the semantics of each application, so these can not be hard-coded in the framework.

Concerning the environment, the system manages a hierarchy of objects of class `MonitoredResource` accessible from a unique `ResourcesManager`. The top-level node, named `'host'`, represents the local machine. Intermediary nodes represent categories (`'storage'` for example), and leafs represent individual resources (for example, `'hda'` representing the first hard drive). Each of these nodes is described through *attributes* attached to them, which are simple *name* / *value* pairs. The values of these attributes can change dynamically and hence represent the current state of the resource they are attached to. Figure 5 summarizes this organization.

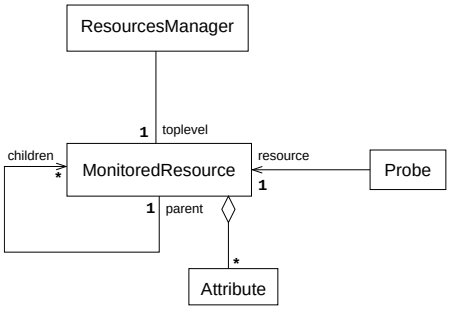

Fig. 5. `MonitoredResource`s and `Attribute`s

A particular resource can be designated using a syntax similar to Unix paths: `'/host/storage/permanent/hda'` designates the node representing the first hard drive. Accessing the value of an attribute is as simple: `'/host/storage-`

`/permanent/hda.total_capacity_kb'` represents the capacity of the hard drive. The attributes attached to individual resources are called *primitive attributes*. Another kind of attributes, called *synthesized attributes* can be attached to intermediary nodes. Their values are the result of arbitrary computations over the attributes attached to sub-nodes. This allows for example to summarize information in order to get a higher-level view of the system. Currently, this requires the creation of a subclass of `MonitoredResource` in order to implement the necessary processing in Java.

The exact structure of the tree and the values of the node attributes are managed by instances of class `Probe`. Probes are active objects responsible for gathering information about the execution condition (for example the current load, battery status...). The information reflected by the `MonitoredResource` hierarchy is updated asynchronously by the different probes, and is always available to the adaptation engine through a query interface. Furthermore, `MonitoredResource`s implement a notification protocol, so that the adaptation engine can ask to be notified when a given attribute value changes or when a resource appears or disappears (which would happen for example if a new USB device is hot-plugged or unplugged in a computer).

This system gives the adaptation engine a dynamic view of the current execution context, but this is not enough to allow adaptation; the engine also needs a good knowledge of the program to be adapted itself. As we said earlier in section 4, the components constituting the application are represented indirectly in our system by generic `Container`s. To allow different components to be distinguished by the adaptation engine, we use the same technique as for resources: arbitrary attributes can be attached to containers to describe their underlying component. The system defines a small set of generic attributes which are added automatically to every container, like the `className` attribute, whose value is the name of the class of the underlying component. It is also possible to tell the system to automatically "export" as attributes the fields of the underlying component. Container attributes support the same access protocols as resources attributes (explicit query and change notification).

6 Decision

This section describes how the adaptation engine works, and how it is configured.

The adaptation engine is the heart of our system. It is the link between the two other functionalities: observation and action. When the observation framework detects a meaningful change in the execution environment, it notifies the engine, whose role is to decide what action is necessary to adapt the application, and then to implement this action using the operations made available by containers (attachment or detachment of roles).

The adaptation engine is designed to be completely independent of an application domain. However, adaptation decisions are highly dependent on the nature and semantics of the applications, and on the environments they execute on: a banking application does not have the same constraints as a multimedia

one for example. Thus, the engine must be "specialized" for each application; this is done by configuring the engine using *adaptation policies*. Actually, adaptation policies can be thought of as adaptation programs written in a specific language. The adaptation engine is then an interpreter for these programs. Our long-term goal is for the language(s) used to write adaptation policies to be as high-level and declarative as possible, in order to ease the work of application programmers and administrators.

We decided to split adaptation policies in two parts: one designed to be written by application programmers and free from low-level considerations, and the other more oriented towards system administrators. The goal is to simplify the work of applications programmers, and to allow them to ignore as much as possible platform-specific issues. These issues can be dealt separately by specialists. This split results in two kinds of adaptation policies, expressed at different levels of abstraction: low-level *system policies* and higher-level *application policies*, both of which are complementary and necessary to configure the adaptation engine.

6.1 System Policies

System policies are low-level policies, whose role is to define adaptation rules independently of application semantic. They are normally written by system specialists and – in theory at least – they can be reused in different applications. Concretely, a system policy is a named set of rules of the form *condition* $\Rightarrow$ *actions*, where the *condition* is a boolean expression concerning the execution environment (as reflected by the resources hierarchy described in Sect. 5), and *actions* is a list of actions, each describing the attachment or detachment of a particular role of a service (as described in Sect. 4). The adaptation engine uses the notification interface of the resources hierarchy to listen to the appropriate conditions. When such a condition occurs, it evaluates the set of corresponding actions.

In the current prototype, the policies are written using a Scheme-like syntax (easy to parse and flexible). The language is not a complete Scheme however, and the primitives available to define the condition part of a rule include only basic numerical operations and comparisons, string comparison and logical operators. As for the second part of the rule, it is not directly interpreted as an action *per se*. It actually tells the adaptation engine that a particular role must be attached (with appropriate parameters) or detached. When the rule is activated, the engine will determine if this implies an attachment, a reconfiguration or a removal of the mentioned role. For example, if the rule says that role `some.role` must be attached and that it already is, the engine does not need to do anything (except perhaps a reconfiguration if attachment parameters differ).

Figure 6 shows four different (but related) system policies, which could be used in our bookstore example application to manage server-side and client-side distribution. The first two are generic server-side and client-side policies, with only one rule each which are always activated. Their corresponding action is to attach a `distribution.rpc.server` (resp. `distribution.rpc.client`) to

functional objects, enabling simple remote message calls. The third policy is designed to be used by clients with a wireless network connection. Because this kind of connection tends to provide highly variable bandwith, the policy adds a role implementing asynchronous method calls with transparent futures; this way the client is never blocked by a sudden drop in bandwidth, enhancing the user experience. The second rule adds a `smart-proxy` role (local caching of remote objects fields) to save bandwidth when it becomes too low[2]. Finally the last policy is to be used by mobile hosts relying on a battery: when the autonomy drops below 5 minutes, we add a persistence role to make sure we will not loose data in case of power failure.

```
(def-policy "distribution.server"
  (when #t (ensure-attached "distribution.rpc.server")))

(def-policy "distribution.default-client"
  (when #t (ensure-attached "distribution.rpc.client"
                            ((server . "pollux.info.emn.fr")))))

(def-policy "distribution.wireless-client"
  (when (= (attr "/host/network.connection-type") "wireless")
    (ensure-attached "messaging.asynchronous-with-future"))
  (when (and (= (attr "/host/network.connection-type") "wireless")
             (< (attr "/host/network.available-bandwidth_kbs") 5)
    (ensure-attached "smart-proxy"))))

(def-policy "distribution.mobile-client"
  (when (and (not (attr "/host/battery.charging"))
             (< (attr "/host/battery.autonomy_s") 300))
        (ensure-attached "persistence")))
```

Fig. 6. Examples of a system policies

In our example application, the first policy could be installed on the enterpise server to enable remote access to the catalog. The second one would be installed on a customer's desktop computer, while the second, third and fourth would all be deployed on his laptop. When the laptop is used as a desktop, using AC power and connected on the LAN, the rules defined in the `distribution.wireless-client` and `distribution.mobile-client` are inactive, and both client hosts behave in the same way. When at the end of the day, the customer takes his laptop home, our monitoring framework detects the

[2] If the bandwidth varies a lot around 5 kb/s, this can lead to unstable states where the service is constantly attached and detached. This can be solved using less naive conditions, for example relative to synthesized attributes implementing a hysteresis mechanism.

change and the system activates the appropriate rules to adapt client-side distribution.

6.2 Application Policies

Application policies are higher-level than system ones and, as their name imply, are designed to be written by programmers specifically for an application (*i.e.* they are not reusable). System policies answer two questions: *When?* (the condition part of the rules) and *What?* (the action part). However, they do not say *to whom* to apply these actions. This is the role of application policies: they tell the adaptation engine which of the functional components present in the system must be affected by which system policy.

To fulfill this goal, application policies define two things:

1. *components groups*, designating "similar" functional components, that must be adapted the same way;
2. and *bindings* between these groups and previously defined system policies (or specific services).

As we said earlier when describing components and containers (see Sect. 4), the only things that can be used to distinguish functional components are the attributes attached to their containers. Groups are hence defined according to the values of these attributes. More specifically, a component group is defined by three things: a name, a *super-group* and a filtering predicate expressed over components attributes. The members of the group are then all the members of the super-group for which the predicate evaluates to true. A special group, named `all` and containing every functional component in the system is used to "bootstrap" this definition mechanism. Of course, components attributes are dynamic, and can appear, disappear or change during the program execution. The system uses the notification interface of component attributes, combined with incremental evaluation of the predicates in order to track the dynamic composition of all the groups.

For each of these dynamic groups, the application policy then defines bindings to system policies. These provide the missing link between system policies and application policies. The semantic is simple: when a system policy is bound to a group, every adaptation rule in this policy applies to the members of this group. Instead of binding a whole system policy, it is also possible to bind a specific role, including configuration parameters for the role.

Figure 7 shows examples of application policies which could be used with our bookstore example application. The first two policies are intended to be deployed on the server to complement the system policies already presented. The first one simply identifies the functional components which will be affected by the `distribution.server` system policies (and hence be accessible remotely). The second policy defines a dynamic group of functional components, consisting in all the `Order`s of more than $ 500. These components are bound to a `logging` policy, not presented here but which logs all the activities on these components. The

group it defines, `logged-components`, is dynamic in the sense that its filtering predicate depends on the `totalAmount` attribute, which reflects a field in the underlying `Order` Java object, and hence can change dynamically.

The third policy, `remote-components.desktop`, binds the default `distribution.default-client` to the functional components, making them simple proxys for the components living on the server. The last policy is almost the same, but is intended for laptops, and adds the corresponding system polcies we described earlier.

```
;; Enterprise Server
(def-group "distributed-components" :parent "all"
  (select (or (= (attr "className") "com.mycompany.Catalog")
              (= (attr "className") "com.mycompany.Customer")
              (= (attr "className") "com.mycompany.Book")
              (= (attr "className") "com.mycompany.Order")))
  (bind "distribution.server"))

(def-group "logged-components" :parent "all"
  (select (and (= (attr "className") "com.mycompany.Order")
               (> (attr "totalAmount") 500)))
  (bind "logging"))

;; Desktop Client
(def-group "remote-components.desktop" :parent "all"
  (select (or (= (attr "className") "com.mycompany.Catalog")
              (= (attr "className") "com.mycompany.Customer")
              (= (attr "className") "com.mycompany.Book")
              (= (attr "className") "com.mycompany.Order")))
  (bind "distribution.default-client"))

;; Laptop Client
(def-group "remote-components.laptop" :parent "all"
  (select (or (= (attr "className") "com.mycompany.Catalog")
              (= (attr "className") "com.mycompany.Customer")
              (= (attr "className") "com.mycompany.Book")
              (= (attr "className") "com.mycompany.Order")))
  (bind "distribution.default-client" "distribution.wireless-client"
        "distribution.thin-client"))
```

Fig. 7. Example of an application policy.

6.3 Interpreting adaptation policies

Once both kinds of adaptation policies are loaded into the adaptation engine, they are merged together in a common internal model easier to interpret (see figure 8).

For each group, a `GroupBindings` object is created. Then, for each system policy bound to this group (in the application policy), each rule it defines is translated into a `Condition` and a `ServiceBinding` (including optional parameters). The resulting *conditional binding* is managed by the `GroupBindings` object.

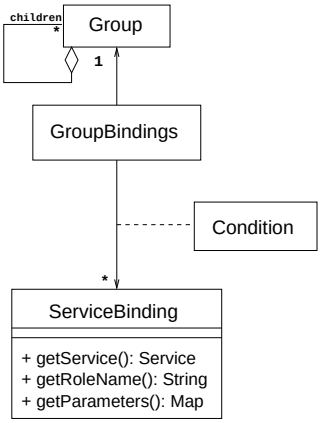

Fig. 8. Internal model used by the adaptation engine

The `Group` itself is only responsible to track its content (which functional components enter and leave it). The `GroupBindings` is the heart of the adaptation engine: it is responsible to implement the conditional binding for all the elements in the group, and only them. Concretely this means that at every instant, all the components currently in the group must have attached to them all the roles defined by the active bindings (the ones whose `Condition` is currently `true`).

To do this, the `GroupBindings` registers as a listener both from the group itself and from the condition. When it detects that a condition's value changes, it attaches (if it became `true`) or detaches (if it became `false`) the corresponding role to/from every component currently in the group. Symetrically, when the `Group`'s content changes, all the currently active bindings are attached to the new components and detached from the ones who leave the group.

The performance and threading issues are not handled in the current model and prototype: detection of changes and modification of the system happen asynchronously and the platform offers no guarantee on the order of the operations. Lots of other issues remain unhandled currently, like detection and correction of conflicts between policies, and guarantees that components reconfigurations do not interfere with their normal execution. All these issues are part of our planned future works, and the current implementation focused on providing a simple proof of concept.

7 Prototype Usage

Concretely, our current prototype of this architecture consists in two elements: a special compiler used to make the application classes reflective [7] and an application launcher. The application launcher is fed a set of configuration files, including system and application policies, and the name of the main program to run. Its task is to setup the probes (part of the monitoring framework, Sect. 5), to discover the available services, and to configure the adaptation engine with the supplied policies. When this is done, it simply starts the application program.

From the point of view of developers, probes and services can be developed independently of any application (and of each other), by system specialists. Some general system policies can also be developed and provided with the base platform. The application programmer only has to develop the business objects required for his application and to compile them with the provided tool which builds reflective classes. Then, he must determine which non-functional services he wants for his components and encode this decision in an application policy. The rest of the development can be handled by someone else (a deployer or administrator). His job is to deploy the application on a specific platform. He needs to get probes and services implementations appropriate for this platform, and then reuse system policies or write custom ones.

Using loose coupling between all the elements of our system (functional and non-functional components, system and application policies), our architecture allows a strong separation of the different roles: each of these tasks can be handled separately by specialists.

8 Related Works

This section discusses some of the other research projects in the field of adaptable middleware, and their relations to our work.

Open-ORB [9, 4] is a project whose goal is to define an architecture for adaptable middleware, particularly in the domain of multimedia applications. It is based on a component model which includes explicit dependencies and interaction between components, support for continuous media (streaming), and an event notification model. This model is used both for base-level components and for a structured meta-level components (called *meta-space models*). Adaptation is done through specialization of components at the meta-level, affecting the behavior of base-level components. Open-ORB has a lot of similarities with our model, but it does not put the same emphasis on how the adaptation decisions are taken (they must basically be programmed "by hand").

DART [10] is a platform allowing the development of distributed applications. It uses reflection techniques to enable dynamic adaptation, both at the base level and at the meta level, using mechanisms similar to a *Strategy* pattern [11] where the method lookup mechanism is extended to take into account the external execution conditions. The mechanisms it provides are very sophisticated and powerful, but actually using these mechanisms requires explicit programming of the adaptation policies.

The dynamicTAO ORB [5] is an extension of the CORBA compliant TAO ORB. TAO is a free ORB designed in a very modular way, so that it is possible to build (statically) custom versions of the ORB by choosing among different implementations for each part of the ORB engine. dynamicTAO makes it possible to use these reconfiguration capabilities at runtime. It uses a *Strategy* pattern [11] to organize the available implementations of ORB elements, and a monitoring service can be used to switch at runtime between these implementations, based on the changing execution conditions. The main problem with this approach is that the changes are system-wide; it is not possible to adapt independently different parts of the system.

MolèNE [12] is a framework designed to ease the development of adaptable applications in the domain of large scale mobile systems. It uses a reflective architecture where the adaptation code is written at the meta-level. When MolèNE's *Detection and Notification Framework* (similar to ours but more sophisticated) detects a change in this environment, it notifies the *adaptive components* constituting the application. These components are able to switch at runtime between different implementations (suited to different execution conditions). The adaptation strategies are encoded in an automata associated to adaptive components. The MolèNE framework is quite complete and sophisticated, but it handles only functional adaptation, and it requires multiple implementations of every functional components.

In [13] is described an architecture designed to ease the adaptation of applications to changing execution conditions. The model considers different layers (operating system, middleware, application, and user), each of which is described using metadata in order to ease their collaborations. When the application invokes a service, the middleware uses both the application metadata and the metadata reflecting the execution conditions to decide how to implement this service. Applications can also ask the middleware to be notified when specific execution conditions occurs. This system allow fine adaptation of applications, but it requires that services call are coded explicitly in the applications. Although we agree with the authors that complete transparency is not possible if we want adaptation (which requires awareness), we tried with our approach to untangle the core application code (not adapted directly in our system) from non-functional services (our main subject of adaptation).

9 Conclusion

In this paper, we presented an infrastructure which can be used to render middleware platforms dynamically adaptable relatively to changing execution environment (Sect. 3). It uses a simple observation framework to reify the state of the execution environment and of the program itself, and meta-programming techniques to modify the application programs by dynamically attaching or detaching non-functional services to functional components. Its main advantage over existing solutions is that all the adaptation code is expressed in *adaptation policies* using two special-purpose languages which are interpreted at run-time

by an *adaptation engine*. One of these languages is low-level and designed to be written by system administrators and deployers. The other is higher-level and designed for application programmers. Loose coupling between the elements of our system enables a strong separation of the different roles in application development, and the potential for reuse of code like probes and services.

9.1 Future Works

Future works are planned to improve the individual parts of our system, but without modifying the general architecture. The adaptation policies and the adaptation engine executing them will evolve toward a higher level of abstraction, with more declarative languages. We also want to study the relationships between our approach and Aspect-Oriented Programming and aspect languages [14].

A global formalization of the model and of its execution semantics – including concurrency – will also need to be developed. Concerning performance, we will investigate the use of partial evaluation techniques at different stages in the development process; for example, it should be possible using tools like AspectJ to statically weave non-functional aspects with functional components at deployment-time if we can detect (analyzing adaptation policies) that some bindings will never be changed. Finally, an important research direction will concern how multiple instances of our system can collaborate in a large-scale distributed application, where no one has a complete control or knowledge over the services and policies used by other parties.

References

[1] DeMichiel, L., Ümit Yalçinalp, L., Krishnan, S.: Enterprise JavaBeans^{TM} Specification. SUN Microsystems. (2001) Version 2.0, Final Release.

[2] Object Management Group: Common object request broker architecture (CORBA/IIOP), version 2.5. OMG Document formal/2001-09-01 (2001)

[3] Sessions, R.: COM+ and the battle for the Middle Tier. Wiley (2000)

[4] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran, H., Parlavantzas, N., Saikoski, K.B.: A principled approach to supporting adaptation in distributed mobile environments. In: 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000), Limerick, Ireland (2000)

[5] Kon, F., Romàn, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: Proceedings of Middleware 2000, International Conference on Distributed Systems Platforms, New York, USA. Volume 1795 of LNCS., Springer-Verlag (2000) 121–143

[6] Bouraqadi-Saädani, N.M.N., Ledoux, T., Südholt, M.: A reflective infrastructure for coarse-grained strong mobility and its tool-based implementation. In: Invited presentation at the *International Workshop on "Experiences with reflective systems"* (held in conjunction with Reflection 2001). (2001)

[7] David, P.C., Ledoux, T., Bouraqadi-Saâdani, M.N.: Two-step weaving with reflection using AspectJ. In: OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, USA (2001)

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001. Volume 2072 of LNCS., Springer-Verlag (2001) 327–353
[9] Andersen, A., Blair, G.S., Eliassen, F.: OOPP: A reflective component-based middleware. In: NIK 2000, Bodø, Norway (2000)
[10] Raverdy, P.G., Lea, R.: Reflection support for adaptive distributed applications. In: Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC '99). (1999)
[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Professional Computing Series. Addison-Wesley (1994)
[12] Malenfant, J., Segarra, M.T., André, F.: Dynamic adaptability: the MolèNE experiment. In Yonezawa, A., Matsuoka, S., eds.: Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, AITO, Springer-Verlag (2001) 110–117
[13] Capra, L., Emmerich, W., Mascolo, C.: Reflective middleware solutions for context-aware applications. In Yonezawa, A., Matsuoka, S., eds.: Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, AITO, Springer-Verlag (2001) 126–133
[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Volume 1241 of LNCS., Springer-Verlag (1997)