A Framework for Monitoring and Reconfiguration of Components Using Dynamic Transformation

Djamel Belaïd*, Imen Ben Lahmar*, and Hamid Muktar†
*Institut Telecom; Telecom SudParis, CNRS UMR SAMOVAR, Evry, France
Email: {djamel.belaid, imen.ben_lahmar}@it-sudparis.eu
†National University of Sciences and Technology (NUST), Islamabad, Pakistan
Email: hamid.mukhtar@seecs.edu.pk

***Abstract*—Distributed applications can be created using component-based software development. Such applications are defined as an assembly of components requiring services from and providing services to each other. The existing component models provide a description of functional and non-functional requirements of an application. However, this capability is to be determined at the design time of the application. Once deployed, the application cannot be modified to respond to the changing context.**

In order to allow creation of such applications that can be transformed dynamically to respond to changing environments, in this article we propose a framework that allows monitoring and dynamic reconfiguration of different components. These components may be functional components of the user application or other components of the environment on which an application depends. The components of environment may represent the underlying environment (i.e., hardware and network entities) and are presented in our framework in the same way as the application components. A component can monitor other components in order to be aware of their changes. Moreover, the components can also be monitored and reconfigured remotely. If a component is not monitorable or reconfigurable by default, we propose a procedure that transforms it to respond to components requests.

***Keywords*-Framework, component model, monitoring, reconfiguration, UPnP, transformation**

I. Introduction

With the emergence of wireless technologies and ubiquity of hand held wireless devices, application development for the pervasive environments is gaining more and more attention. In such environments, computing is pushed away from the traditional desktop to small hand-held and networked computing devices that are present everywhere we go. As such, Service-Oriented Architecture (SOA) has emerged as a computing paradigm that changed the traditional way of how applications are designed, implemented and consumed in a pervasive computing environment.

One particular approach for developing SOA-based applications is to use component-based application development. Using this approach, an application is defined as a composition of re-usable software components, which implement the business logic of the application domain collectively by providing and requiring services to/from one another. The components required by an application are assembled at the time of application development. Thus, at the time of application deployment, all the components have been defined statically.

However, when considering the broad range of computing devices in pervasive environments (smartphones, PDAs, tablets, laptops, etc.) – with different capabilities and limitations - this approach may not work. Moreover, pervasive environments are highly dynamic due the mobility of users and devices. Thus, an important aspect of pervasive applications is that their realization is very much dependent on their execution context. Due to variability of the environment, modelling the application behaviour needs to satisfy not only the functional requirements in an effective way, but in order to provide better quality of service (QoS) for user satisfaction, it should also consider the current state of the environment in which the application is executing. In addition, the application should also adapt itself according to changing context.

Existing component models like PCOM [1] [18], Fractal [4], OSGi [17] and SCA [5] propose application development using component assembly. In these models, a component offers its capabilities through provided interfaces and consumes functionalities offered by other components through required interfaces. Along with offered and required interfaces, a component may define one or more properties. These properties can also be modified so that a component can be reconfigured dynamically at runtime.

Due to the heterogeneity of the environment, modelling the application behaviour needs to consider the current state of the environment in which the application is executing in addition to its functional requirements. However, most of the existing component models leave such issues to the underlying middleware, which provides a uniform Application Programming Interface (API) or a framework for this purpose [1] [18]. This means that the programmers and the designers have to rely on the functionality of the underlying middleware and such aspects need to be considered during application development life-cycle. All of the reconfiguration aspects, such as determination of reconfigurable properties, have to be decided at development time. Once an application has been developed, and deployed, its composition becomes fixed. A property that was not set to be reconfigurable or monitorable during development remains so and changes need to be made at source code level to make it reconfigurable or monitorable dynamically.

In order to explain these limitations, let's consider an application that provides the functionality of sending large

files from one device to another using a communication link that exists between them. Files are transferred via a WiFi connection and as the size of a typical file is large (e.g., 1 GB each), each file is transferred in several compressed chunks (e.g., 256 KB) to allow quick transfer. On the receiving side, once a device receives a compressed chunk, it decompresses it before merging it with other chunks to reconstruct the whole transferred file.

Due to variability of the WiFi signal strength, the application needs to continuously monitor the network signal to decide the chunk size to be used for sending the file. In case of high signal strength, data can be sent at higher rates and larger chunk size can be used; however, in case of weak signal strength, a smaller chunk size may be applied for a quick transfer. Moreover, to decide the degree of compression, the application needs to monitor the remaining battery powers of the sender and receiver devices. If the battery power of any of the devices is low, it uses lower degree of compression to conserve the battery power required for the compression and/or the decompression. However, if the remaining battery power is sufficiently high for both devices, higher degree of compression can be used for a better throughput over the network. As it can be seen, the decompression degree depends on the used degree of compression. Thus, the use of a lower/higher compression degree for a given file chunk at the sender's side implies that the decompression degree is to be maintained the same on the receiving side. Therefore, whenever the compression ratio changes, the decompression degree should be reconfigured for an efficient transfer of files over the network.

We deduce a few important points from the file sender application. First, the behaviour of the application is dependent on certain properties which are not part of the File Sender application, namely, remaining battery power and network signal strength. Both of them correspond to externally required properties: they do not form the core logic of the application, however, the desired Quality-of-Service (QoS) provided by the application greatly depends on their values. Thus, a mechanism is needed to consider such required properties in the application design without altering the application logic and architecture. Second, the monitoring can be notifications from the provider side or observations from the client side. Third, the reconfiguration of components' properties is dependent on changes of properties of other components. Thus, we require a mechanism that is able to reconfigure dynamically the properties of components whenever there is a need. Finally, since these properties may belong to components found locally or remotely, we need a uniform strategy for accessing local or remote properties to be monitored or reconfigured.

As discussed previously, the above mentioned requirements are not considered by the existing component models. Therefore, in our previous work [2], we have proposed a component model that 1) considers explicitly the required properties of an application in addition to the functional behaviour; 2) allows the monitoring ; and 3) if a property is not monitorable by default, we provide transformation mechanism to render it monitorable.

In this article, we propose the following contributions, some of which extend our previous work [2]. As an extension, first, we present a remote monitoring approach allowing components to monitor their remote required properties. Second, we extend our proposed component model to allow components to reconfigure their local and remote required properties. Third, we present some transformation processes for components to render their properties monitorable or reconfigurable whenever there is a need by a third-party.

The remaining article is structured as follows. In Section II, we first describe our proposed component model and the component assembly, then, in section III, we explain the monitoring and reconfiguration concepts and how components can be transformed to make them monitorable or reconfigurable locally as well as remotely. Section IV describes how components can be declared and how the transformation can be achieved followed by the implementation details in section V. In Section VI we provide an overview of existing related approaches as well as their limitations. Finally, Section VII concludes the article with an overview of our future work.

II. Our Approach

In this section, we outline the different concepts involved in our approach. We begin by introducing our component model and component assembly. We then describe the need for component transformation and how they are accompanied by integrating adaptive logic into the application to make it adaptive to the changing context.

A. The Component Model

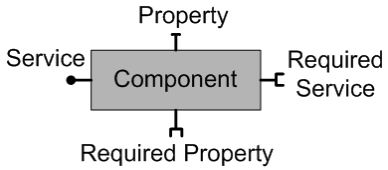


Fig. 1. Component model describing required properties

In an object-oriented paradigm, an object provides its services through a well-known interface, which specifies the functionality offered by the object. In the relatively newer component-oriented paradigm, components may specify, in addition to their provided interfaces, their dependencies on the offered properties of other components using required interfaces. As defined by Szyperski et al. [20] "A software component is a unit of decomposition with contractually specified interfaces and explicit context dependencies only." Thus, a component not only exposes its services but it also specifies its dependencies. Most of the existing component models [1][4][17][5] allow specification of their dependencies for business *services* external to the component. However, they do not allow specification of their dependency for external *properties*. The ability to specify dependency for external properties has two important implications. First, it results

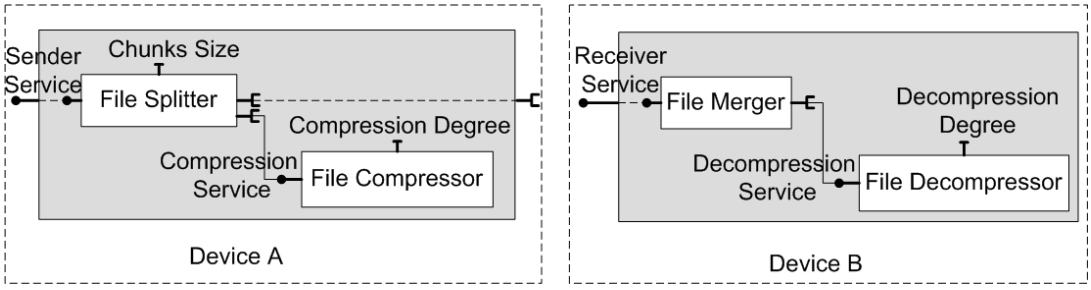


Fig. 2. File Sender Application

in specification at relatively fine granularity thus helping the architects and designers in fine tuning the component's requirements. Second, this fine tuning helps in elaborating the contract between two components because the properties can be enriched with additional attributes that constrain the nature of the contract through appropriate policies. To achieve this objective, in one of our previous works [2], we have proposed a component model that allows expressing this dependency explicitly in terms of required properties, which are provided by other components.

Figure 1 shows main characteristics of a component that provides a service through an interface and requires a service from other components through a reference. The component also exposes a property through which it can be configured. In addition, the component also specifies its dependency on a certain property. This required property, which appears at the bottom of the component, will be satisfied if we can link this component with another component that offers the requested property, thus, solving the dependency.

We use components to represent not only software entities that make up an application, but also to represent hardware and network entities present in the execution environment. For example, a component may represent the screen of a device, a WiFi card, or even user preferences.

B. Component Assembly

Components can be combined together in an assembly of components to build complex applications. For example, figure 2 shows how the File Sender application described in the introductory section can be represented by an assembly of components distributed across two devices: two of them on the sender device (A) and two others on the receiving device (B).

On the sender side (device A) the File Splitter component splits a given file into chunks for an efficient transmission. The appropriate size of the file chunk is determined by the network signal strength. If the signal strength is high, data can be sent at higher rates and larger file chunks (e.g., 1 GB each) will be created. However, if the signal is weak, smaller chunks (e.g., 256 KB) will be created to allow quick transfer. Once a file chunk is created, it is passed to the File Compressor component for compression before sending it to the receiver device. This is done using the service provided by the File Compressor component. The File Compressor component uses an adaptive compression algorithm whose compression ratio depends upon the remaining battery powers of the sending and receiving device. If the remaining battery power of each device is above a certain threshold (e.g., 20 percent), higher degree of compression is used. However, if the remaining battery power of any of the devices is below the specified threshold, lower degree of compression is used by doing quick compression of each chunk thereby conserving the battery power.

On the receiving side (device B), a File Decompressor component is used to decompress the received compressed chunk. The component has a decompression degree property whose value should be the same as the value used for compression by the File Compressor component. Thus, any change of the compression degree must imply the same change of the decompression degree in the File Decompressor component. Once the received chunk is decompressed, a File Merger component combines it with the other decompressed chunks to recreate the transferred file.

C. Component Transformation

Figure 2 shows the File Sender application as defined by the architect. As can be seen, it represents only the functional components of the application and does not show the components external to the application — battery and WiFi — which are required by the functional components for providing the necessary QoS desired by the user. Assume that this application was developed for fixed environments, e.g. a desktop PC connected to a wired network with fixed QoS, in which the application would not need to be adapted.

Our objective is that given such an application, which was not conceived for dynamic environments with changing QoS, we would like to transform it in order to adapt its functionality according to the changing QoS. This transformation can be of two types. If we are interested in knowing the changes in the properties of a component, we need to transform the component to make it monitorable. On the other hand, if we are interested in modifying the properties of a component due to external notifications, we need to make it reconfigurable. Furthermore, a transformation may apply to a component available on the device locally resulting in local transformation or it may apply to some component in remote device, in which case it is known as remote transformation.

In our previous work [2], we proposed a monitoring mechanism to permit an application — based on our component

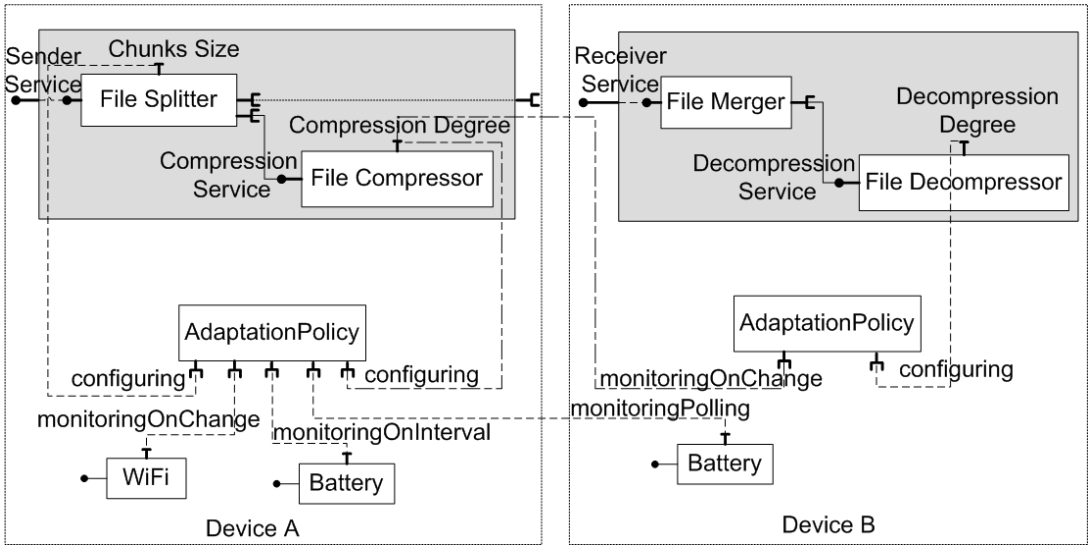


Fig. 3. Monitoring and Reconfiguration of required properties of the File Sender Application

model — to be adapted. However, the approach was limited to monitoring in the local scope. In our present work, we go a step further to address the remote monitoring as well as the local and remote reconfiguration of the components.

Corresponding to our scenario, our application needs to monitor the components of environment: battery and WiFi. Assuming that these components are not monitorable by default, we need to make them monitorable by transformation. Similarly, the functional components of the application need to be reconfigured every time the QoS provided by these non-functional components changes. The File Splitter component needs to be reconfigured for its chunk size property whenever the WiFi signal strength changes. The compression degree of the File Compressor component should also be reconfigured whenever the battery level crosses its threshold. Corresponding to File Compressor component, the File Decompressor component must also be reconfigured with the same degree of decompression as that used for compression.

Since the components are not reconfigurable by default, i.e., when they were defined initially during application assembly, we need to transform them to make them reconfigurable. In the next section, we describe how these transformations can be used along with some adaptation policies to render an application adaptable.

D. Adaptation Logic

Transformation allows a component to be monitorable or reconfigurable. However, only transforming an application does not help in making the application adaptive. For example, in our example application, only by making the WiFi component monitorable and the File Splitter component reconfigurable will not make the application to take adaptation decisions. Instead, we need some *adaptive logics* that will make appropriate adaptation decisions based on certain rules for adaptation. This adaptation logic has to be defined by the architect at design time to make the application adaptable. It is encapsulated in an adaptation policy component which is a functional component and defined following our component model. The adaptation policy components express through their required properties their need to monitor and/or to reconfigure local as well as remote properties offered by components of the adaptable application.

III. Monitoring and Reconfiguration Framework

To make our example application adaptable, we transform its components to make them monitorable or reconfigurable and integrate an adaptation policy component that defines the policy to be used for adaptation. The transformed File Sender application is shown in figure 3. Two Adaptation Policy components have been used to manage the adaptation of the application to respond to adaptation due to the change of the context. The context is defined by the properties of the WiFi and the Battery components and the adaptation is made on the properties of the File Splitter, File Compressor and File Decompressor components. These adaptation policy components express their need to monitor and to reconfigure local and remote properties of other components through their required properties.

On device A, the adaptation policy component expresses through its required properties its need to monitor the *signal strength* property of the WiFi component, the *level* property of the local Battery component and the *level* property of the remote Battery. Depending on changes in the values of these properties, the Adaptation Policy component also expresses its need to reconfigure the *chunk size* property of the File Splitter component and the *compression degree* property of the File Compressor component.

On device B, another adaptation policy component is used to monitor remotely the *compression degree* property changes of the File Compressor and to reconfigure the *decompression degree* property of the File Decompressor component accordingly.

```
public interface GenericProxy {
  Property[]  getProperties();
  Object  getPropertyValue(String propertyName);
  void  setPropertyValue(String propertyName,
                         Object propertyValue);
  Object invoke(String methodName, Object[] params);}
```

Fig. 4. Description of the Generic Proxy interface

If the offered properties of components are not defined as monitorable or reconfigurable resources, we need to transform them to respond to the requests of the adaptation policy components. A transformation is applied dynamically at runtime and is carried out by some predefined components of our framework. For different types of transformation, the framework has defined different components. In the next subsections, we introduce them and we detail the main features of the monitoring and the reconfiguration mechanisms and their transformation processes.

A. Generic Proxy Service

The Generic Proxy Service, provided by our framework, can be applied to any component of an application that we want to introspect before any transformation.

We have defined a general purpose interface GenericProxy that provides four generic methods. These methods are described in figure 4. Each implementation of this interface is associated with a component for which the first method getProperties() returns the list of the properties of the component, the getPropertyValue() returns the value of a property, the setPropertyValue() changes the value of a property and the invoke() method invokes a given method on the associated component and returns the result.

We provide two implementations of the GenericProxy interface (see section V). The first one, LGenericProxy component is for implementation of a local proxy. That is, when associated with a local component it translates its method calls into calls of the associated component. The second one, RGenericProxy component is a remote implementation. That is, when associated with a remote component it forwards the calls of its methods to calls of the associated remote component, over the network.

B. Local Reconfiguration

In parametric adaptation, a component is able to reconfigure the properties of another component. In this context, reconfiguring the required properties of a component is defined as the reconfiguration of the offered properties of another component. For this purpose, we extend our component model [2], in order to allow components to specify their need to reconfigure some of their required properties. For example, in figure 5(a), a component A specifies a required property, offered by the component B, that it needs to reconfigure.

A given property of component can be reconfigured by calling its associated setter method. However, the component that wishes to reconfigure a property of another component does not know a priori the type of this component. To complete the reconfiguration of any component from only the name and type of a property, the reconfigurator component uses an appropriate interface that provides the method **setPropertyValue(propertyName, propertyValue)** to change the value of a property.

However, the component to be reconfigured may not define its properties as reconfigurable resources despite the request. So we need to transform the component to make its properties reconfigurable by offering an appropriate reconfiguration interface. This can be done dynamically by our framework by encapsulating the component with the predefined LGenericProxy component as defined above. The two components are combined together in a single *composite* that offers the services of the original component as well that of the LGenericProxy component. The component can be then reconfigured using the setPropertyValue() method provided by the LGenericProxy component. The framework then replaces the original component with the newly created composite in the application. Figure 5(c) shows the transformation of the component B to render its property reconfigurable by the component A.

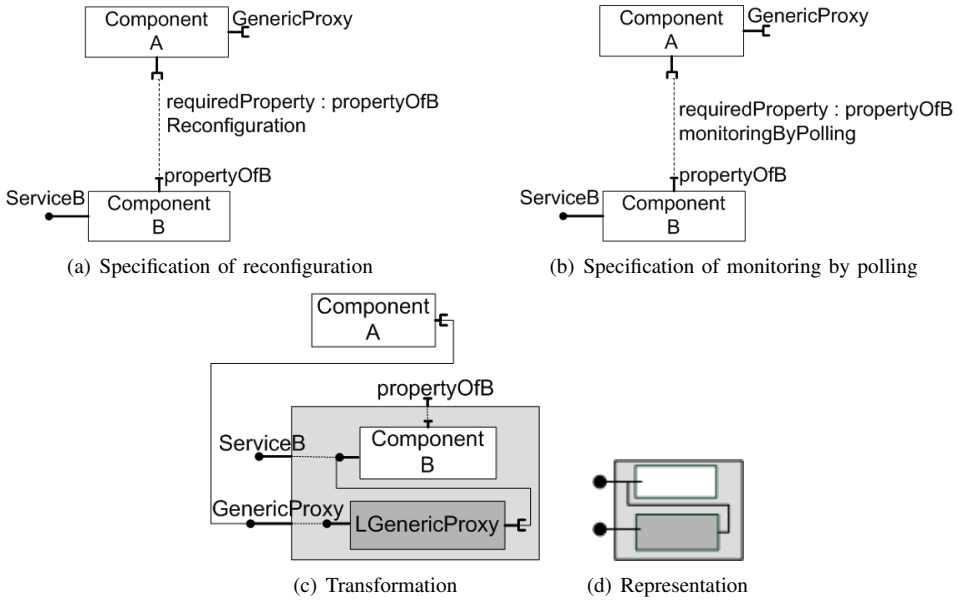
C. Local Monitoring

In [2], we have presented a monitoring approach to allow a component to be aware of required properties changes. Monitoring process consists in informing the interested component about the changes of required properties or notifying it on a regular way or for each variation. We have considered two types of monitoring: monitoring by polling and monitoring by subscription.

Polling is the simpler way of monitoring, as it allows the observer to request the current state of a property whenever there is a need. However, subscription allows an observing component to be notified about changes of monitored properties.

1) Monitoring by Polling: A component may express its need to monitor by polling a required property provided by another component (figure 5(b)). The monitoring by polling of a property can be made by calling its getter method. However, the component that wishes to monitor a property of another component does not know a priori the type of this component. To complete the monitoring of any component from only the name and type of a property, the interested component often uses an appropriate interface that provides the method **getPropertyValue(propertyName)** to request the current state of a property.

However, the component to monitor may not define its offered properties as monitorable by polling resources despite the request. So, we need to transform the component to make its properties to be monitorable by offering an appropriate interface of monitoring. This can be done dynamically by our Framework at runtime in the same way as the transformation process for the reconfiguration since it uses the same LGenericProxy component that provides the needed



(a) Specification of reconfiguration (b) Specification of monitoring by polling

(c) Transformation (d) Representation

Fig. 5. Monitoring by polling and reconfiguration

interface. In figure 5(c), we show the transformation of the component B to render its property monitorable by polling by the component A. Figure 5(d) is a symbol, we used to represent the transformation for monitoring by polling as well as reconfiguration.

2) Monitoring by Subscription: There are two modes of monitoring by subscription: 1) subscription *on change* which specifies that the subscribed component is notified every time the value of the property changes; 2) subscription *on interval* which specifies that the subscribed component is to be notified after a specified time interval. For notification on change, a component must precise the starting time and the duration of notifications. For notification on interval it must specify the notification interval value. It may also precise the starting time and the duration of notifications. The component A must also implement a notification callback through which it will receive appropriate notification messages.

Figure 6(a) shows how component A specifies its need to monitor a required property offered by a component B by subscription on change. Figure 7(a) shows monitoring by subscription on interval. Figures 6(c) and 7(c) shows the symbols we used for denoting subscription on change and subscription on interval, respectively.

For the monitoring with notification mode on interval, as shown in the figure 7(b), each time the MonitoringBySubscription component have to notify the subscriber (the component A), it gets (or monitor by polling) the value of the required property of the component B via the LGenericProxy component.

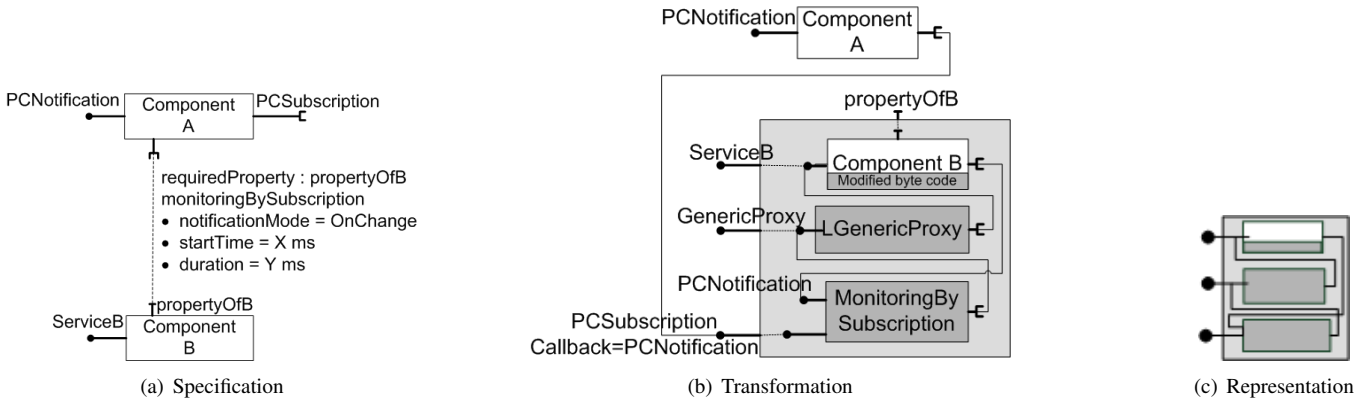
When the notification mode is on change for a required property of B (figure 6(b)), the MonitoringBySubscription component offers a (callback) service of notification PCNotification to the component B so that it can be notified of the changes of a required property and in turn inform all the subscribers of this change. To allow the component B to notify the MonitoringBySubscription for the change of its properties, our framework adds the needed instructions in the byte-code of the component B at runtime.

D. Remote Monitoring and Reconfiguration

To adapt a distributed application in a pervasive environment, some components may be interested in monitoring or reconfiguring properties of other components remotely. For this purpose, components in our framework can specify their need about remote reconfiguration or remote monitoring of some of their required properties.

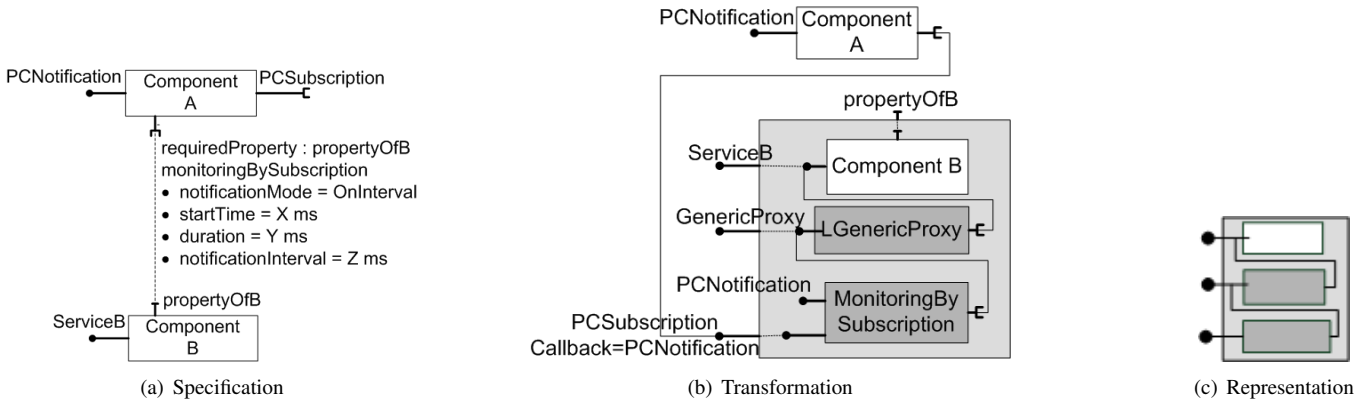
For example, figure 8 shows how component A specifies its need to monitor by subscription a property offered by a remote component B. For the two modes of notification, the component B (the server) must offer a remote subscription service over the network to the component A (the client) and in turn, the component A must subscribe to the remote component B specifying its need. When a change of the property happens, a notification from the component B to A is sent over the network. As for the local case, to provide a remote reconfiguration and monitoring by polling the component B must offer a GenericProxy service and should also be reachable over the network. We note that for the remote purposes there are two transformations: server-side (component B) and client-side (component A).

1) Server-side transformation: The framework first encapsulates the component B in a composite (figure 9(a)) such as defined for the transformations for the monitoring by subscriptions. Then it adds a new component (the RServer component) that integrates the network communication aspects like remote call and event processing. The RServer component has two references: one for the subscription service offered the MonitoringBySubscription component and one for the Gener-



(a) Specification (b) Transformation (c) Representation

Fig. 6. Monitoring by subscription with notification mode on change



(a) Specification (b) Transformation (c) Representation

Fig. 7. Monitoring by subscription with notification mode on interval

icProxy service provided by the LGenericProxy component. The first reference is used to subscribe and the second one is used for the reconfiguration and monitoring by polling on behalf of the client component A. We note that the newly defined composite provides the GenericProxy service and the PCSubscription service in addition to the services of the component B. Thereby it can be used, reconfigured and monitored locally as well as remotely. Figure 9(b) is a symbol denoting the server-side transformation.

2) Client-side transformation: If the component B was on the same device as the component A, we would have used the same composite defined for the monitoring by subscription and connect A to its subscription interface. Since this is not the case, our framework creates the same kind of that composite (figure 10(a)) with the ServerProxy component in place of the component B and for the implementation of the interface GenericProxy we use the RGenericProxy described in section III-A. The ServerProxy component is a byte-code generated component by our framework at runtime (see section V). Its implementation of the service B consists on forwarding the calls to the RGenericProxy component which in turn make the call over the network to the server side. Figure 10(b) is a symbol, we used to represent the transformation in the client-side.

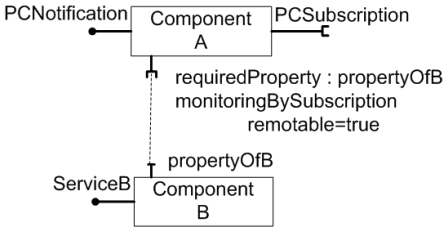


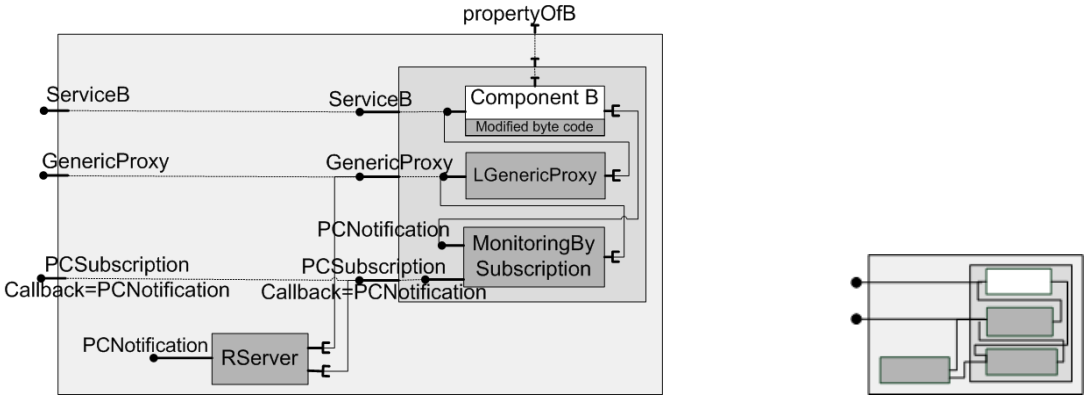
Fig. 8. Specification of remote monitoring

E. Putting it all together

Referring back to figure 3, we assume that signal strength, the compression degree and the local and remote battery levels are not defined as monitorable properties. So we need to transform WiFi, File Compressor and the two Battery components to render their properties monitorable by the Adaptation Policy components.

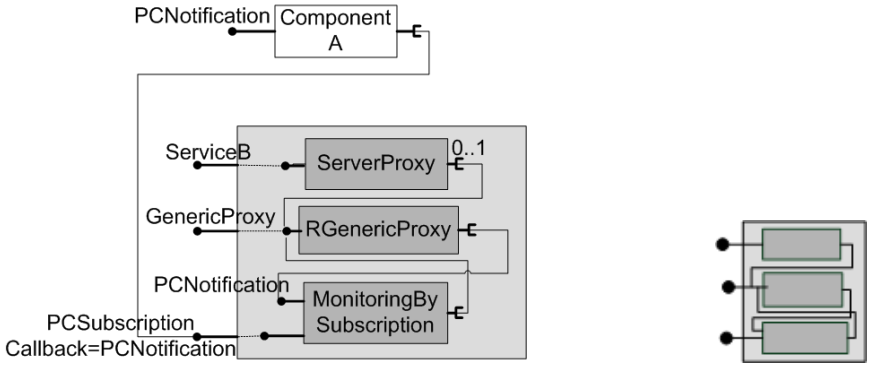
Figure 11 shows the creation of new composites in response of the monitoring request of the Adaptation Policy components. The transformation of the WiFi component corresponds to the creation of a composite as shown in figure 6(c), while the local Battery component is transformed following the figure 7(c).

The Battery component on device B is remotely observed



(a) Transformation (b) Representation

Fig. 9. Remote monitoring and reconfiguration: server side



(a) Transformation (b) Representation

Fig. 10. Remote monitoring and reconfiguration: client side

by the Adaptation Policy component of device A. For this purpose, we require two transformations: a server-side (Battery component) and a client-side (Adaptation Policy component) transformation. The transformation of the Battery component is done following figure 9(b), and a new composite representing the remote Battery component is created in the device A following figure 10(b) to allow the Adaptation Policy component to observe the battery level changes.

Similarly, the File Compressor component (server side) is transformed following the figure 9(b) to allow the Adaptation policy component of device B (client side) to subscribe to the changes of the compression degree property.

For the reconfiguration needs of the Adaptation Policy components, our framework transforms the File splitter and the File Decompressor components following the figure 5(d). The File Compressor component is already transformed into a composite that offers a generic proxy service for reconfiguration.

IV. Architectural Description

The description of an application can be done with the help of an Architecture Description Language (ADL). Instead of inventing a new ADL, we prefer to use one of the existing description languages. In this regard, Service Component Architecture (SCA) [5] provides a rich ADL that details most of the aspects that we are looking for. One of the main features of this component model is that it is independent of particular technology, protocol, and implementation. It consists of a set of services, which are assembled together to create solutions that serve a particular business need. These services correspond to offered services and required ones called references. Along with services and references, a component can also define one or more properties. We use SCA for its extensibility to overcome the missing elements related to required properties, monitoring and reconfiguration aspects.

In our previous work [2], we have extended the standard SCA description by adding the `@requiredProperty` attribute to express explicitly the dependency of components to the offered properties of other components. This attribute specifies the resource whose property will be monitored or reconfigured.

Moreover, we have also specified the `resource` type of each component or property. For a component, the resource type corresponds to the classification of the component in one of the predefined categories, such as software, hardware, and network, etc. Some of these categories have been defined previously as extension of the Composite Capability/Preference Profile (CC/PP) [14] by the authors [16]. For a property,

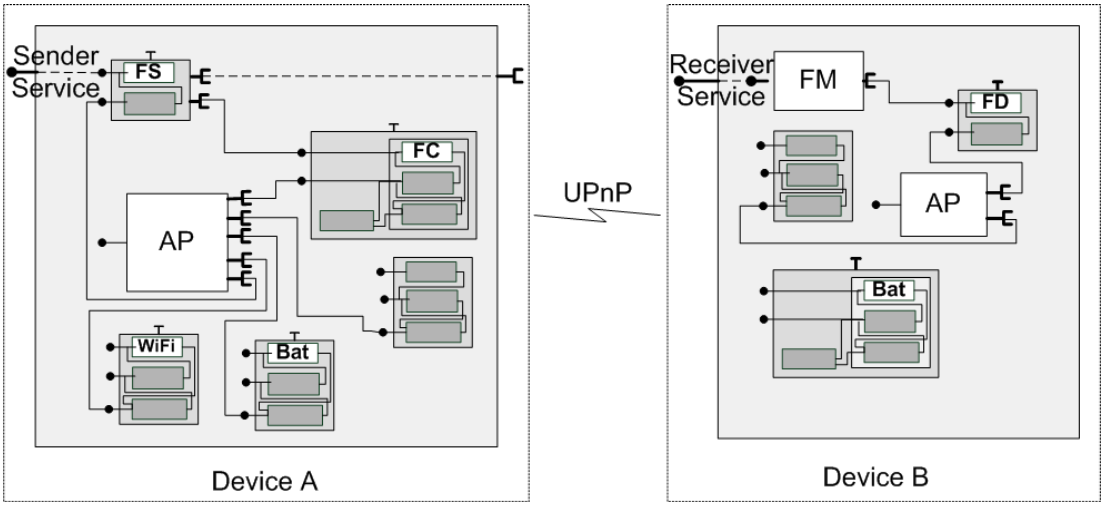


Fig. 11. Transformation of the File Sender Application

```
1<composite name="FileSender">
2   <service name="FileSenderService" promote="FileSplitter/FileSenderService"/>
3    ........
4   <component name="AdaptationPolicy" resource="Software.Component">
5      <service name="PCNotification">
6         <interface.java interface="eu.tsp.iaria-example.PCNotificationInterface"/>
7      </service>
8      <requiredProperty resource="Hardware.Battery" remotable="true" monitoring="ByPolling">
9          <property name="BatteryLevel"/>
10      </requiredProperty>
11     <requiredProperty resource="network.WiFi" monitoring="BySubscription" notificationMode="ON_CHANGE">
12         <property name="SignalStrength"/>
13     </requiredProperty>
14     <requiredProperty resource="Software.FileSplitter" reconfiguration="true">
15         <property name="chunckSize"/>
16     </requiredProperty>
17          ....
18   </component>
19   <component name="FileSplitter" resource="Software.Component">
20     .......
21   </component>
22 </composite>
```

Fig. 12. Description of the File Sender Application using our extended SCA ADL

the *resource* type specifies the component to which this property belongs. The separation between property name and the associated resource type is significant as it allows the transformation of a given component into a monitorable or reconfigurable component for only the specified properties.

Figure 12 shows the description of the File Sender application. As it can be seen (lines 11-13), a `@monitoring` annotation is used to specify the subscription requirement for `SingalStrength` property. Moreover, a `@notificationMode` annotation is used to specify the monitoring by subscription mode. The Adaptation Policy component requires the monitoring of the `SingalStrength` property of the WiFi component, belonging to the network category, by subscription with notification mode *on change* .

In addition to the monitoring feature, the required property allows components to specify their configuration requirements. For this goal, we have extended the `@requiredProperty` attribute to express explicitly the configuration requirements. The extension consists of a new `@reconfiguration` annotation that specifies if the property of resource would be reconfigured or not. For example, the AdaptationPolicy component requires the reconfiguration of the chunckSize property offered by the File Splitter component (lines 14-16).

To handle remote components by monitoring or configuring their offered properties, we used the `@remotable` annotation of SCA specification [5] to specify if the required property is remotable or not. The `@remotable` annotation has boolean value; if it is true, it implies that the requested resource is remotable else it is a local resource. For example, the Battery component (lines 8-10) provided by device B, is remotely

```
1<composite name="FileSender">
2   <service name="FileSenderService" promote="FileSplitter/FileSenderService"/>
3     ........
4   <component name="AdaptationPolicy">
5      <service name="PCNotification">
6         <interface.java interface="eu.tsp.iaria-example.PCNotificationInterface"/>
7      </service>
8      <reference name="PCSubscriptionService" target="WiFiComposite"/>
9      <reference name="PCSubscriptionService" target="BatteryComposite"/>
10     <reference name="PCSubscriptionService" target="UPnPClientComposite"/>
11     <reference name="GenericProxyService" target="FileSplitterComposite"/>
12     <reference name="GenericProxyService" target="FileCompressorComposite"/>
13   </component>
14   <component name=FileSplitterComposite>
15     <service name="FileSenderService">
16         <interface.java interface="eu.tsp.iaria-example.FileSenderInterface"/>
17     </service>
18     <implementation.sca name="FileSplitterComposite"/>
19  </component>
20</composite>
```

Fig. 13. Transformation of the Adaptation Policy component

```
1<composite name=FileSplitterComposite>
2  <service name="FileSenderService" promote="FileSplitter/FileSenderService" />
3  <service name="GenericProxy" promote="LGenericProxy/GenericProxy" />
4  <reference name="FileCompressorService" promote="FileCompressor/FileCompressorService" />
5  <reference name="FileReceiverService" promote="FileSplitter/FileReceiverService"/>
6  <component name="FileSplitter" resource="Software.Component">
7      <service name="FileSenderService">
8         <interface.java interface="eu.tsp.iaria-example.FileSplitterInterface"/>
9      </service>
10     <implementation class="eu.tsp.iaria-example.impl.FileSplitterImpl"/>
11     <reference name="FileCompressorService" target="FileCompressorComposite/FileCompressorService" />
12     <reference name="FileReceiverService" target="FileMerger/FileReceiverService" />
13  </component>
14  <component name="LGenericProxy" resource="Software.Component">
15     <service name="GenericProxy">
16         <interface.java interface="eu.tsp.iaria-example.GenericProxy"/>
17     </service>
18     <implementation class="eu.tsp.iaria-example.impl.LGenericProxy"/>
19     <reference name="FileSplitterService" target="FileSplitter/FileSenderService"/>
20  </component>
21</composite>
```

Fig. 14. Transformation of the File Splitter component

monitorable by polling by the Adaptation Policy component.

Figure 13 shows the transformation of our extended SCA description of the File Sender application into standard SCA description. As it can be seen, the required properties of the Adaptation Policy component are transformed into references to the created composites (lines 8-12).

In figure 14, we show the transformation of the File Splitter component to a new composite to render its property reconfigurable by the Adaptation Policy component. The created composite exposes in addition to the **FileSender** service of the File Splitter component, a GenericProxy service that is provided by a Local Generic proxy component allowing the reconfiguration of the chunckSize property.

V. IMPLEMENTATION

In order to validate our approach, we have implemented a prototype of our approach in Java. The prototype implements our framework as services that offers the various transformation mechanisms to the applications.

For dynamic transformation we required code level manipulation for which we used the open source software JAVA programming ASSISTant (Javassist) library [13]. Javassist is a class library for editing byte codes in Java; it enables Java programs to define a new class at runtime and to modify a class file when the Java Virtual Machine (JVM) loads it. The implementation of the **LGenericProxy** component implements the GenericProxy interface (figure 4). It is based on the Java reflection API. As defined in the java API specification [12],

the java API java.lang.reflect provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use reflected fields, methods, and constructors to operate on their underlying counterparts on objects.

For remote communication, we have used the Universal Plug and Play (UPnP) [21] technology. A UPnP network consists of UPnP devices that act as servers to UPnP control points, the clients. The control point can search for devices and invoke actions on them. The **RServer** component, as shown in figure 9 (a), integrates the network communication aspects like remote method call and event processing on behalf of the server component it represents and is implemented as a UPnP device. For that purpose, it uses the services (interfaces) of the server component, generates a UPnP device and services (actions and stateVariables) descriptions and starts the device to be detected (by the UPnP control points) over the network. When it receives from control points a call as a UPnP action, it translates it as a call to the appropriate method of the LGenericProxy component which in turn calls the associated method of the server. When it is notified, by the **MonitoringBySubscription** component, for a change of one of the server properties it modifies the related state variable and then the device sends an event over the network with the new value of the property so the control points that has subscribed to that change receives this notification.

The **ServerProxy** component, in figure 10 (a), is used in place of a server when this last is on another device. Thanks to the javasist API, it is Java generated implementation of the services (interfaces) of the server. The calls to its methods are forwarded to the appropriate method of the component, implementing the GenericProxy interface, it references.

The **RGenericProxy** in figure 10 (a), is implemented as an UPnP control point. When it starts, it searches for a UPnP device with the same type of RServer component and subscribe to the UPnP events related to the change of the variables state of this server component. Each time it receives a state variable change event it notifies the MonitoringBySubscription component that in turn notifies the interested subscribers (e.g. ServerProxy component). The RGenericProxy also implements the GenericProxy interface. Each call to a method of this interface is transformed as an UPnP action call to the device server (i.e. RServer component).

Finally, to allow a component to notify the MonitoringBySubscription for the change of its properties, the open-source Javassist is used, at runtime, to inject the notification code in the property setter of the byte-code of the component implementation (class).

VI. Related Work

The monitoring and reconfiguration issues have been extensively studied in different contexts, notably in the area of software components that has become an accepted standard for building complex applications. In this section, we detail some of the existing related approaches as well as their limitations. But let's first give an overview of some component models that become an accepted standard for building complex applications.

The OSGi component model [17] enables components to hide their implementations from other components while communicating through services that are shared between components. It allows specification of properties when registering components in the service registry. This ensures searching and binding of components having specific properties. An advantage is that these properties cover a wide range of characteristics such as context, quality of service, and other non-functional aspects. However, the major drawback is that the specification of properties is done at the code level; so properties are tied to the functional code.

Another relevant component model is SCA [5], which provides a programming model for building applications and systems based on a Service Oriented Architecture. More recently, there has been SCA extension for event processing [6]. It is based on the publish/subscribe model, which allows components to produce and consume events through channels. These events may be sent to notify a consumer about changes occurring on the producer's side, without the consumers having knowledge of the producer's functionality. Currently, the SCA specifications of existing runtimes do not provide any transformation mechanism to render a property nor monitorable neither reconfigurable.

The PCOM component model [1][18] allows specification of distributed applications made up of components. Components reside within a component container that contains listeners allowing application programmers to be notified whenever a parameter or a communication changes or new components are discovered. Each component explicitly specifies its dependencies in a contract, which defines the functionality offered by the component, i.e., its offer, and its requirements with respect to local resources and other components. To model the required properties, the syntactical description can be enriched with properties of typed name-value pairs for offers and typed name-value-comparator triples for requirements. Using this description, the system can automatically determine whether an offer can satisfy a requirement.

In a recent work [10], the PCOM container was extended by two services; assembler and application manager, to separate the task of calculating the configuration from the application execution. The application manager is responsible for managing the life cycle of application and also the selection of the configuration algorithm. However, the assembler is responsible for calculating a valid configuration. Clearly, using a fully distributed assembler, for instance, requires the availability of an instance of this assembler on each device.

One of the limiting factors of PCOM resource description is that resources are not standardized: there is no formal way of resource description and the different types of resources can not be distinguished from one other. Another problem is that the non-functional aspects of an application are also treated in the programming API. This means the component developer has to take care of the non-functional aspects (monitoring,

binding, and unbinding of the components) at the code level. Moreover, PCOM does not support the remote monitoring of components' parameters, since each container is reponsible of the monitoring of the hosted components.

In [3], Beugnard et al propose a process that makes functional aspects of components independent from observational ones. This separation of concerns allows the advantage of changing observations without modifying the core part of components. They have also defined a set of predefined components dedicated to observation that can be attached to any functional component. Both the functional and observation components are defined declaratively and then using a kind of weaver they can be integrated to result in context-aware components.

While their approach is quite general, we can identify two limitations compared to our proposed approach. First, instead of defining a few dedicated observation components, we propose that any of the components in our system can be transformed into observation component by adding to it the capabilities of observation. Second, since this transformation is done dynamically, we can selectively specify the properties of a component that we want to make observable.

As a monitoring and a reconfiguration middleware, we cite MADAM (Mobility and ADaption enAbling Middleware) [8], which is a middleware for runtime adaptation with: context management, adaptation management and configuration management. Their objective is to adapt the application at runtime in response to context changes. For this purpose, the middleware provides a Context Manager to monitor context when this latter changes. It is responsible for context reasoning, such as aggregation, derivation, and prediction in order to provide the Adaptation Manager component with relevant context information when context changes occur. The Configurator middleware component is responsible for reconfiguring an application by deleting or replacing component instances, instantiating components, transferring states, etc.

However, MADAM middleware does not support the remote monitoring and the remote reconfiguration and it is limited to the local scope. Moreover, it does not support the transformation of components to render them monitorable or reconfigurable resources.

In the same context, we cite the AwareWare middleware that tends to facilitate the development of applications to be more adaptive in such a heterogeneous environment [22] [23]. The AwareWare middleware consists of Awareness measurement tools that are used to measure and to collect awareness data. The awareness manager organizes these tools and provides system independent query and notification interfaces for adaptive applications. They consider two basic methods, pull and push, for querying distributed awareness information. An adaptation decision module and adaptation policy language are used to reconfigure applications. The adaptation policy defines rules to determine how the application changes its behaviours, by changing the applications component inter-connections and tuning parameters.

However, the reconfiguration is limited to the hosted components and it does not cover the remote components. Moreover, AwareWare middleware does not support the transformation of components to monitorable of reconfigurable resources. However, our framework is able to transform the components into monitorable or reconfigurable resources to reply to the components' request.

In another recent related work [7], a reconfiguration middleware handles the reconfiguration of applications in heteregenous environment. Towards this objective, Corradi et al propose to partition the middleware logic into two reconfiguration layers that basically feature an application logic layer and a non-functional layer on top of a very minimal kernel layer. Each reconfiguration layer features a monitoring engine whose aim is to keep track of current status of elements of the (monitored) layer above. A reconfiguration enactment engine concretely executes the reconfiguration actions determined by policies.

Compared to our approach, the major drawback of the reconfiguration middleware that does not support the remote monitoring and the remote reconfiguration of applications. Further, the proposed middleware does not consider the transformation of components to monitorable or reconfigurable resources despite the components' request.

The approaches in [9] and [11] focus on the adaptation of system behaviour at runtime. They consists of reusable infrastructure corresponding to probes, and resource discovery components to support respectively, monitoring of properties changes, and quering for new resources. An adaptation engine is used to carry out the necessary reconfiguration by using some adaptation operators and adaptation strategies.

However, the reconfiguration of components does not cover the remote components and it is limited to the local scope. Moreover, they do not support the transformation of components to render them monitorable or reconfigurable by other components. However, in our approach, we propose to model explicitly these features through required properties of components, and even these latter are not defined by default as monitorable or reconfigurable resources, our framework is able to transform them to reply to components request.

In [15], Melisson et al propose pervasive binding to provide support for service discovery in SCA-based applications. This binding is called UPnP binding since it is based on UPnP protocol [21]. Towards this objective, they propose to integrate UPnP into FRASCATI platform (a SOA platform for the SCA standard)[19] to support the remote call of a service that is advertised via the UPnP protocol. The FRASCATI platform was modified to supply spontaneous communications between SCA components using UPnP bindings.
However, in our approach, the SCA composite is transformed by adding UPnP components allowing its remote monitoring and its remote reconfiguration using the existing open-source SCA runtimes (e.g. Newton, Tuscani, etc.). Moreover, their approach is limited on the remote call and it does not take into account neither the remote monitoring nor the remote reconfiguration. However, in our present work, we rely on UPnP protocol to manipulate remote components by monitoring their

changes and reconfiguring their properties in addition to the remote call.

Project	Monitoring		Reconfiguration	
	Local	Remote	Local	Remote
Madam Middleware [8]	√		√	
Rainbow Middleware [9]	√		√	
AwareWare Middleware [22] [23]	√	√	√	
Reconfiguration Middleware [7]	√		√	
PCOM Component Model [1][10]	√		√	√
SLCA Component Model [11]		√	√	
Our Framework	√	√	√	√

TABLE I
EXAMPLES OF MONITORING AND RECONFIGURATION APPROACHES

In table I, we summarize the most important features related to some of the cited middlewares and component models. We compare them regarding the monitoring level, i.e., if it is executed locally or remotely, and the similarly the reconfiguration level. As it can be seen, most of the given approaches focus on the local monitoring and/or local reconfiguration mechanisms. There are limited approaches that consider remote monitoring and/or remote reconfiguration. And even if they exist, their presented mechanisms are not defined in appropriate way. Moreover, in our knowledge, none of the cited middleware supports the transformation of components to render them monitorable or reconfigurable resources.

VII. CONCLUSION AND FUTURE WORK

In this article, we proposed an approach for monitoring and dynamic reconfiguration of component-based applications. The flexibility offered by our approach is that any software, hardware, or network component that one wants to monitor or reconfigure, but that does not offer these capabilities inherently, can be transformed to offer these functionalities — given that they are representable by a software component. These aspects are treated independently of the functional code and, hence, do not make the situation more complex for the designers and developers.

For this purpose, we proposed a generic component model that allows unified specification of hardware, software, and network components. In our component model, a component specifies its provided and required services, and its properties, but in addition, it also specifies the required properties that are to be monitored or reconfigured. If the required properties are not monitorable or reconfigurable by default, transformation processes are done dynamically by our framework to reply to the component request. Some Java implementation details of the prototype were provided in the article.

We are integrating our prototype into an SCA runtime. This will allow us to test the feasibility of our approach in real-world scenarios.

VIII. ACKNOWLEDGMENTS

This work is partially supported by French ANR through Seamless (SEamless and Adaptive Services over MultipLe AccEsS NetworkS) project number 07TCOM018.

REFERENCES

[1] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - A Component System for Pervasive Computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, PERCOM '04, pages 67–76, Orlando, Florida, USA, 2004.

[2] Imen Ben Lahmar, Hamid Mukhtar, and Djamel Belaïd. Monitoring of non-functional requirements using dynamic transformation of components. In *Proceedings of the 6th International Conference on Networking and Services*, ICNS'10, pages 61–66, Cancun, Mexico, 2010.

[3] Antoine Beugnard, Sophie Chabridon, Denis Conan, Chantal Taconet, Fabien Dagnat, and Eveline Kaboré. Towards context-aware components. In *Proceedings of the first international workshop on Context-aware software technology and applications*, CASTA '09, pages 1–4, Amsterdam, Netherlands, 2009.

[4] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice and Experience (SP&E)*, 36:1257–1284, 2006.

[5] Open SOA Collaboration. SCA Assembly Model Specification V1.00. http://www.osoa.org/, 2007.

[6] Open SOA Collaboration. SCA Assembly Extensions for Event Processing and Pub/Sub V1.00. http://www.osoa.org/, 2009.

[7] Antonio Corradi, Enrico Lodolo, Stefano Monti, and Samuele Pasini. Dynamic reconfiguration of middleware for ubiquitous computing. In *Proceedings of the 3rd international workshop on Adaptive and dependable mobile ubiquitous systems*, ADAMUS'09, pages 7–12, London, United Kingdom, 2009.

[8] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23:62–70, March 2006.

[9] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, October 2004.

[10] Marcus Handte, Klaus Herrmann, Gregor Schiele, and Christian Becker. Supporting pluggable configuration algorithms in pcom. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, PERCOMW'07, pages 472–476, NY, USA, 2007.

[11] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, Mobility'08, pages 11:1–11:8, Yilan, Taiwan, 2008.

[12] Java 2 Platform API Specification. http://download-llnw.oracle.com/javase/1.4.2/docs/api/java/lang/reflect/package-summary.html.

[13] JAVA programming Assistant. http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[14] Cédric Kiss. Composite capability/preference profiles (cc/pp): Structure and vocabularies 2.0. w3c working draft 30 april 2007. http://www.w3.org/TR/2007/WD-CCPP-struct-vocab2-20070430/, 2007.

[15] Rémi Mélisson, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. Supporting Pervasive and Social Communications with FraSCAti. In *3rd DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*, CAMPUS'10, Amsterdam, Netherlands, 2010.

[16] Hamid Mukhtar, Djamel Belaïd, and Guy Bernard. A policy-based approach for resource specification in small devices. In *Proceedings of the second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 239–244, Valencia, Spain, 2008.

[17] OSGI. Open services gateway initiative. http://www.osgi.org, 1999.

[18] Stephan Schuhmann, Klaus Herrmann, and Kurt Rothermel. A framework for adapting the distribution of automatic application configuration. In *Proceedings of the 5th international conference on Pervasive services*, ICPS'08, pages 163–172, Sorrento, Italy, 2008.

[19] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA applications with the frascati platform. In *Proceedings of IEEE International Conference on Services Computing*, SCC'09, pages 268–275, Bangalore, India, 2009.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Boston, MA, USA, 2nd edition, 2002.

[21] UPnP Forum. UPnP Device Architecture 1.1. http://www.upnp.org, 2008.

[22] Qinag Wang and Liang Cheng. Awareware: an adaptation middleware for heterogeneous environments. In *IEEE International Conference on Communications*, Paris, France, 2004.

[23] Qinag Wang and Liang Cheng. A flexible awareness measurement and management architecture for adaptive applications. In *IEEE Global Telecommunications Conference*, GLOBECOM'04, Dallas, Texas, USA, 2004.