An Approach for Applications Suitability on Pervasive Environments*

Andres Flores[1] and Macario Polo[2]

[1] GIISCo Research Group
Departamento de Ciencias de la Computación, Universidad Nacional del Comahue,
Buenos Aires 1400, 8300, Neuquen, Argentina

[2] Alarcos Research Group
Escuela Superior de Informática, Universidad de Castilla-La Mancha,
Paseo de la Universidad 4, 13071, Ciudad Real, Spain

Abstract. This work is related to the area of Component-based Software Development, particularly to largely distributed systems as Pervasive Computing Environments. We are focused on the automation of a Component Integration Process as a support for run-time adjustments of applications when the environment involves highly dynamic changes of requirements. Such integration implies to evaluate whether components may or may not satisfy a given model. The Assessment procedure is based on syntactic and semantic aspects, where the latter involves assertions, and usage protocols. We have implemented on the .Net technology the current state of our approach to gain understanding about the complexity and effectiveness of our approach.

1 Introduction

Pervasive Computing Environments (PvCEnv's) should support the 'continuity' of users' daily tasks across dynamic changes of operative contexts. However functionality is usually shaped as a set of aggregated components which are distributed among different computing devices. On changes of availability of a given device the involved component behaviour still needs to be accessible in the appropriate form according to the updated technical situation. This generally makes users to be involved on a dependency with the underlying environment and increases the complexity of its internal mechanisms [1].

Applications composed of dynamically replaceable components imply the need of an appropriate Integration Process according to Component-based Software Development (CBSD) [2, 3]. For this an Application Model may provide the specification of a required functionality in the form of the aggregation of Component Models A Component Model provides a definition to instantiate a component and its composition aspects through standard interactions and unambiguous interfaces [4, 5]. In order to assure the adequacy of a given component with respect to an Application Model there is a need

* This work is supported by the projects: CyTED-CompetiSoft, UNCo-MPDSbC (04-E059) and UCLM-MAS (TIC 2003-02737-C02-02)

Flores A. and Polo M. (2006).
An Approach for Applications Suitability on Pervasive Environments.
In *Proceedings of the 3rd International Workshop on Ubiquitous Computing*, pages 71-78
DOI: 10.5220/0002504000710078

to evaluate its Component Model. Hence we present an Assessment procedure which can be applied both on a development stage and also at run-time. We compare functional aspects from components against the specification provided by the Application Model, which is component-oriented. Besides analysing component services at a syntactic level, its behaviour is also inspected thus embracing semantic aspects. The latter is done by abstracting out the black box functionality hidden on components in the form of *assertions*, and also exposing its likely interactions by means of the *usage protocol* [6] – also called choreography [7].

We illustrate with a simple example both the way a functionality is composed from distributed disparate components and how the Assessment procedure helps to assure the suitability of a certain involved component. We have implemented on the .Net technology the current state of our approach. The use of certain built-in mechanisms of .Net allow us to retrieve component interfaces and also to incorporate information for evaluation. Though being simple, this prototype give us a rewarding data on possibilities to make concrete our proposals. All the applied techniques are selected according to our goal of achieving consistent mechanisms to assure a fair component integration. As we proceed with our work, reliability is mainly considered, since we focus the whole integration process for those challenging systems as PvCEnv's.

This paper continues by presenting the proposal for an integration process on Section 2. Section 3 illustrates the approach with a simple case study. Section 4 presents the .Net prototype. Conclusions and future work are presented afterwards.

2 Component Assessment for Application Integration

In previous works [8, 9] we have described a preliminary model for the assessment procedure. In this paper we extend the model by adding more aspects concerning semantic information in the form of the abstract behaviour for a component as well as the protocol of use. We assume that a component under evaluation should satisfy a certain degree of compatibility with respect to a given requirement specification. Such specification is assumed as being part of an Application Model which contains the components and describing the required functionality in a component-oriented form by including the following aspects:

1. *Expected Interface*. Signatures of expected services.
2. *Abstract Behaviour*. Assertions for the component and its services.
3. *Usage Protocol*. The expected order of use for its services.

Based on this we make the following consideration upon similarity between components. A component B offers similar functionalities to the expected ones (A) when the three following conditions are properly satisfied:

Condition 1. Component B offers, at least, the same or equivalent services as those offered by A.

$$\texttt{Interface(A)} \subseteq \texttt{Interface(B)}$$

Condition 2. Abstract behaviour of component B is similar or equivalent to component A.[3].

$$\texttt{Behaviour(A)} \approx \texttt{Behaviour(B)}$$

[3] We use $\approx$ to denotes "equivalence", which depends on the element to be compared and the applied technique

Condition 3. The protocol of use for services on both components is equivalent.

$$\texttt{UsageProtocol(A)} \approx \texttt{UsageProtocol(B)}$$

Condition 1 is true when there exists equivalence on corresponding services from both interfaces. This is fulfilled when the five next conditions are satisfied:

Condition 1.1. The amount of services on B is at least the same as in A.
Condition 1.2. The return type of a pair of services sa of A and sb of B is equivalent[4].
Condition 1.3. The number of parameters on services sa of A and sb of B is the same.
Condition 1.4. Parameter types on a pair of services sa of A and sb of B are equivalent.
Condition 1.5. Parameters inside the list of parameters on a pair of services sa of A and sb of B are in the same order.

Condition 2 is true when there exists equivalence on the pre- and post-conditions for corresponding services from both components. Assertions are boolean functions composed of expressions connected by operators $\wedge$ and $\vee$ [10]. A service sa of A is equivalent to a service sb of a component B when the three next conditions are satisfied:

Condition 2.1. Data types included on expressions of assertions of sa and sb are similar.
Condition 2.2. Pre-condition of sb is at most as restricted as pre-condition of sa:
Pre-cond. of sb may have less expressions than in sa. At least one expression on sb's pre-cond. must be equivalent to a corresponding one on sa. $\texttt{pre(sb)} \leq \texttt{pre(sa)}$
Condition 2.3. Post-condition of sb is at least as restricted as post-condition of sa:
Post-cond. of sb may include more expressions than in sa. All expressions on sa's post-cond. must be equivalent to the corresponding ones on sb. $\texttt{post(sb)} \geq \texttt{post(sa)}$

Condition 3 is true when the usage protocol on both components express a similar order for services. We describe usage protocols by means of regular expressions where the operators are concatenation ($\bullet$), alternative ($+$) and iteration ($*$) – the order is actually described by the concatenation operation. The similarity, then, is based on the following conditions:

Condition 3.1. An expression ($+$) on B must be at least as smaller as the corresponding expression ($+$) for A – e.g. ($\texttt{a}+\texttt{b}+\texttt{c}$) from B and ($\texttt{a'}+\texttt{b'}$) from A.
Do not affect equivalence if an extra service from B is described inside an expression ($+$).
Condition 3.2. For all subexpressions into an expression ($\bullet$) on A there are equivalent counterparts in the same order into the corresponding expression ($\bullet$) for B – e.g. ($\texttt{a}\bullet(\texttt{b}+\texttt{c})$) from B and ($\texttt{a'}\bullet\texttt{b'}$) from A.
Condition 3.3. For all subexpressions composed of just one service into a expression ($\bullet$) on B, there are equivalent counterparts in the same order into the corresponding expression ($\bullet$) for A – e.g. ($\texttt{a}\bullet\texttt{b}\bullet\texttt{c}$) from B and ($\texttt{a'}\bullet\texttt{c'}$) from A are not equivalent.

In order to understand the way these conditions are used to distinguish compatibilitiy we present in the next section a simple example where the assessment procedure is applied.

[4] For built-in types, types on sb must have at least as much precision as types on sa have – e.g. compare double w.r.t. integer.

3 Case Study

Suppose we represent a PvCEnv for a Museum, where there could be for example a Tour Guide application to propose different paths according to the user's dynamic choices. When the user enters the museum may carry a computing device (a PDA or a smart phone) and through an automatic detection the device is identified and connected to the environment. Upon each visited art piece (e.g. painting or sculpture) descriptions and information of particular interest to the user is displayed on the PDA or spoken through the phone. Figure 1 shows a likely scenario of the presented case study.

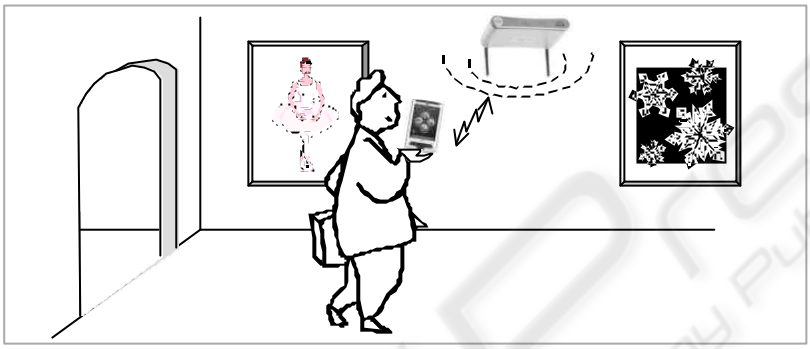

Fig. 1. Likely scenario of a PvCEnv for a Museum.

A related application could allow creating albums with images of the art pieces visited by the user. The Album Organizer application – maybe downloaded into a user's notebook recognized by the environment – may allow creating a sort of document with images and some notes written by the user. Notes could be stored on separated text files and bind to the document by means of hypertext links. Thus every time the user needs to write or edit a note, a proper editor is provided. The user may also be allowed to print a selection of pages of the document, or even send the created album by e-mail to easy carrying those files. We focus on this last application and we analyse its potential required components. There could be an `Album_Organizer` component to represent the main logic of the application, which could have an ad-hoc sophisticated album visual editor or a web-style editor in which is additionally required a generic web browser – the visual editor also depends on the actual used device. For making notes, different components could be used as a simple sort of `NotePad`, `WordPad`, etc – according to the underlying software platform. To send e-mails applications like `Outlook`, `Eudora`, etc, could be used, and to provide a printer service different kind of printers and ad-hoc wireless sensors should be available. Other component is concerned with the data base for images and descriptions of art pieces. Figure 2 shows a diagram with the likely comprised components and devices for the Album Organizer application.

Suppose a user needs to write a note by using a notebook which runs a Linux platform. One available text editor is `KEdit`. The environment then evaluates this component so to ensure it is appropriate to fulfill the task. Following can be seen the interfaces of both the `KEdit` component and the required component model named

Fig. 2. Components for the Album Organizer application.

`TextEditor`. The Assessment Procedure which may provide a degree of compatibility must verify that Condition 1, 2 and 3 are satisfied – as we pointed out on Section 2. We begin analysing Condition 1.

```
component TextEditor {
  void new(string fileName);
  void open(string fileName);
  void save(string fileName);
  void print(string fileName); }
```

```
component KEdit {
  void new(string fName);
  void open(string fName);
  void save(string fName);
  void print(string fName); }
```

3.1 Interface Equivalence

For Condition 1 to be true we verify the five sub-conditions which are related to syntactic aspects. As can be seen both `KEdit` and `TextEditor` include the same amount of services (Cond.1.1), with the same return type (Cond.1.2), the same amount of parameters (Cond.1.3), the same types (Cond.1.4) and in the same order (Cond.1.5). This implies that Condition 1 is satisfied, though it does not give a meaningful evaluation result yet. Every pair of services from both components give an equivalent result. We do not rely on the name of services which could give a difference here. Whether we want to be sure about the utility of the `KEdit` component, a more accurate procedure is still needed. Thus we continue exploring for Condition 2.

3.2 Behaviour Equivalence

Condition 2 is related to the pre and post-conditions from corresponding services. For brevity reasons we describe this procedure only for the `print` service from both components. Assertions are specified by using OCL as follows.

TextEditor
print
pre: `fileName <> BLANK` *and* *not* `printer.queue.Full()`
post: `printer.print(fileName)`

KEdit
print
pre: *not* `printer.queue.Full()` *and* `BLANK <> fName`
post: `printer.print(fName)`

Condition 2.1 is analysed first inspecting data types besides those from the parameter list which have already analysed on Conditions 1.2 and 1.4. In the assertions above can be seen that for the `print` service Condition 2.1 is satisfied. For Conditions 2.2 and 2.3 we derive from assertions Abstract Syntax Trees (ASTs) which we have extended

with the addition of specific features. On each node in the tree we also save a 'type' that is used to operate with its sub-trees: *Interchangeable Operator* (IO) type for values like or, and, $=$, $<>$ etc, meaning that being `a` and `b` two sub-trees, `(a` and `b)` is the same that `(b` and `a)`; *Non-Interchangeable Operator* (NIO) type for values like $>$, $<$, etc; *Unary Operator* (UO) type for values like not, etc – a tree with just one child; *Text* (TXT) type for values being numbers or variable names. Expressions with boolean operators and and or are transformed into a normalized and extended form. For example, `(a` and `(b` or `c))` is equivalent to `((a` and `b)` or `(a` and `c))` but not immediately comparable. Then, the first one is normalized without loosing its semantic. In case of $>=$ and $<=$, they are expanded into two subtrees connected by an or operator – e.g. `(a>=b)` becomes `((a>b)`or`(a=b))`. Figure 3 shows the ASTs for pre-conditions of `print` from both components, from which we start the evaluation procedure. *For this, the root node of both trees are compared and, if they are equal, the respective left and right subtrees are recursively compared*. In our example, both trees present IO root nodes. Thus, we can compare the left sub-tree of one pre-condition with the right sub-tree of the other, and vice versa. This allows to detect the equivalence on both trees. Values on leave TXT nodes are equivalent with respect to their data types (Cond.2.1). Since one tree may have more sub-trees than the other, the extra sub-trees expose that a pre-condition is bigger than the other – as it is described by Condition 2.2. This is not the case for trees on Figure 3, and all the sub-trees are equivalent making pre-conditions being equivalent as well. Similar procedures are followed up for each candidate pair of services in relation to Conditions 2.2 and 2.3. This makes clear the real correspondence on the services from `KEdit` with respect to the expected ones `TextEdit`.

Fig. 3. ASTs for `print`'s assertions.

3.3 Usage Protocol Equivalence

The next step is to check equivalence on the regular expressions describing the protocol of use for a component. The usage protocols for `TextEditor` (1), and `KEdit` component (2) are given below. Usage protocols comparison is also made deriving ASTs as can be seen on Figure 4. The set of operators to set the node types differs based on regular expressions. Concatenation ($\bullet$) is a NIO type. Alternative ($+$) is an IO type. Iteration ($*$) is a UO type. TXT nodes correspond to services in the leaves of the tree, and the equivalence is based on Condition 1 and 2. Thus, as the nodes labelled with $\bullet$ and $+$ correspond to IO operators, the trees can be found equivalent. Therefore, as both Condition 1 and 2 are fulfilled, we can infer that `FinancialAccount` offers similar functionalities to those of `BankingAccount`.

(1) (new+open) • (save+print)* (2) (open+new) • (print+save)*

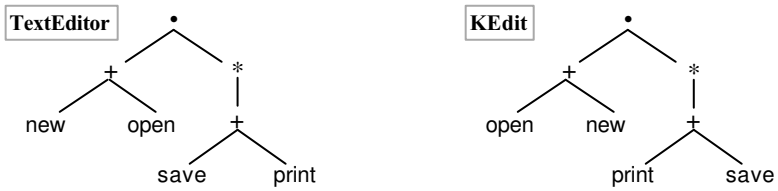

Fig. 4. ASTs for Usage Protocols.

4 Preliminary Implementation

We have developed a first prototype to check the feasibility of our proposal. The prototype is based on Microsoft .NET technology and it includes simple but effective implementations of different elements and algorithms described in the previous section. In order to representing Assertions and Usage Protocol .NET allows to add information to components using the *Attribute* mechanism. This help to annotate classes, methods, parameters, etc. To describe assertions, we have created a class called *Contraint* that specializes *System.Attribute*. This class includes the ambit where the attribute is valid – *Methods* in this case. Each constraint will contain a String representing the text of the pre or postcondition. For regular expressions representing the usage protocol the ambit is *Class*. In order to facilitate evaluation both, the assertions and the usage protocol, are described in a prefix form as can be seen above. In order to inspect the set of members of any element, .NET includes the *Reflection* mechanism. This can be used to retrieve the set of methods from components to be evaluated. Reflection can be of substantial help in cases where components do reside on well-known and already evaluated repositories.

5 Related Work

Research very close to our intent of composing applications is presented in [5], though here we suppose components residing on distributed disparate devices. From this we continue studying technical situations which could make the environment to apply an adjustment over a running application. Particularly the so called quality of service implies an important consideration for this approach. The work in [11] presents a solution for composing applications. A general framework for components integration is presented and evaluated the involved challenges on its application for PvCEnv's. Other work which gives a contribution to our work is presented in [7] by providing a consistent format for specifications of components by means of XML. The approach covers all of the aspects from components: functional, non-functional and commercial. We are evaluating the use of such XML schemas and probably extending those related to non-functional aspects. Some recognized projects on PvCEnv's are Aura and Gaia. The former presented in [12] addresses a large range of PvC topics by focusing of system aspects. Applications are treated as user tasks which are a collection of abstract services and are monitored to optimize their resources. The latter presented in [13] extends the traditional services of an operating system by considering a PvCEnv. Both services and devices are treated as resources that must be managed and allocated to requesting clients.

6 Conclusions

We intend to address the automation of an Integration Process from software components in order to properly update applications into a PvCEnv. In previous works [8, 9], we have presented a scheme to address our intent. In this paper we have explained how components could be replaced when the technical conditions change. This is done according to the Application Model and the connection of Components and Models. We have also described an Assessment procedure to evaluate components both at development stage and at run-time. Such evaluation is based on specifications of the components functionality, which is provided by their Component Models. Compatibility of a component with respect to an expected Component Model is analysed at syntactic and semantic levels. Semantic aspects are described by means of assertions and usage protocols, which are then analysed by deriving extended ASTs – storing both expressions and control data that help in the evaluation process. We have implemented the current stage of our approach on Microsoft .Net in order to gain experience to understand possibilities to recognize not only efficiency but mainly effectiveness on supporting reliability. Selection of appropriate methods, techniques and languages must be accurately accomplished upon the concern of a reliable mechanism. This is the emphasis of our next development in this area.

References

1. Judd, G., Steenkiste, P.: Providing Contextual Information to Pervasive Computing Applications. In: IEEE PERCOM'03, Dallas, USA (2003) 133–142
2. Brown, A., Wallnau, K.: Engineering of Component-Based Systems. In: 2^{nd} ICECCS'96, Montreal, Canada, IEEE Computer Society Press (1996) 414–422
3. Flores, A., Augusto, J.C., Polo, M., Varea, M.: Towards Context-aware Testing for Semantic Interoperability on PvC Environments. In: 17^{th} IEEE SMC'04, The Hague, Netherlands (2004) 1136–1141
4. Heineman, G., Council, W.: Component-Based Software Engineering - Putting the Pieces Together. Addison-Wesley (2001)
5. et.al., B.W.: An Acive-Architecture Approach to COTS Integration. IEEE Software (2005) 20–27
6. Brada, P.: Towards Automated Component Compatibility Assessment. In: 6^{th} Workshop on Component-oriented Programming (ECOOP'01), Budapest, Hungary (2001)
7. Iribarne, L., Troya, J., Vallecillo, A.: A Trading Service for COTS Components. The Computer Journal **47** (2003)
8. Polo, M., Flores, A.: Towards Run-time Component Integration on Ubiquitous Systems. In: 3^{rd} MSVVEIS'05, held during ICEIS'05, Miami, Florida, USA (2005) 9–18
9. Flores, A., Polo, M.: Dynamic Component Assessment on PvC Enviroments. In: 10^{th} IEEE ISCC'05, Cartagena, Spain, IEEE Computer Society (2005)
10. D'Souza, D., Wilis, A.: Objetcts, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley (1998)
11. et.al., T.G.: Pervasive challenges for software components. Technical Report TUV-1841-2003-05, Technical University of Vienna, Vienna, Austria (2003)
12. Page, P.A.H.: Distraction-free Ubiquitous Computing. http://www-2.cs.cmu.edu/ aura/ (2006)
13. Page, G.P.H.: Active Spaces for Ubiquitous Computing. http://gaia.cs.uiuc.edu/ (2006)