Towards Run-time Component Integration on Ubiquitous Systems [⋆]

Macario Polo[1] and Andres Flores[2]

[1] Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, Ciudad Real, España
[2] Departamento de Ciencias de la Computación, Universidad Nacional del Comahue, Buenos Aires 1400, 3400 Neuquén, Argentina

Abstract. This work is related to the area of Ubiquitous Systems and it focuses on a Component-based Integration process. This implies the evaluation of whether components may or may not satisfy a given set of requirements. We propose a framework for such process and consider related concepts like Assessment and Adaptation.
The Assessment procedure is based on meta-data added to components, involving assertions, and usage protocols. Assertions and usage protocols are evaluated by applying a technique based on Abstract Syntax Trees. We also report on a prototype developed to implement the proposed Assessment and Adaptation procedures which allowed us to gain understanding about the complexity and effectiveness of our model.

Category. Work in progress by PhD Student

1 Introduction

An Integration Process for Component-based Systems (CBS) is assumed to enhance reusability by consuming previously developed 'off-the-shelf' (OTS) components [1, 2]. Our main interest is on automating the process as a support for Ubiquitous Systems. The goal on such environments is to provide a feeling of continuity on users' daily tasks. Hence, users do not expect a constrained environment [3, 4], and this can only be achieved by an unlimited availability of resources. Applications are the main resources that must remain available through changes on users context of operation. Some approaches assume a fixed range of applications, or externally developed applications. However, suitability for such applications could be in high risk when the underlying conditions change unfavorably. We assume the possibility to build applications '*on demand*' when necessary. This could be achieved by assembling OTS components [5], and we believe the whole initiative needs the consideration of a more formal framework. We give in this paper a preliminary scheme to address the automation of the whole Integration Process, which concerns '*qualification*', '*adaptation*', '*assembly*' and '*integration*'. We are currently focused mainly on Qualification and Adaptation. In this

[⋆] This work is supported by the CyTED project VII-J-RITOS2, the UNCo MPDSbC project 04-E059 and the UCLM MAS project TIC 2003-02737-C02-02.

paper we present a Component Assessment procedure for qualification, and introduce a preliminar approach for adaptation.

When a required application is not available, the *assembly* procedure could be initiated by previously executing a selection procedure of proper components. They could be fetched from a repository or being discovered from some mobile device. In any case evaluation should be thoroughly done [5, 6]. Our Assessment Procedure intends to compare behavioural aspects from components against a given set of requirements. The requirement specification is assumed in the form of a component interface – the necessary set of component services. Thus our approach, which compares components interfaces, actually evaluates a component against an expected set of services. This implies to evaluate services' signatures at a syntactic level – e.g identifiers, parameters, and data types.

Additionally, components will be enriched by adding meta-data – an adaptation mechanism called *instrumentation* [2]. Thus we intend to embrace semantic aspects. The first concern is adding '*assertions*' which help to abstract out the black box functionality hidden on components. Also, it is included the '*usage protocol*' for services which use regular expressions to describe the expected order of use for component services. This technique has been applied on inter-class testing [7], and also on descriptions for components [8].

Suppose post-conditions (for example) on services from two similar components. They should relate to a similar structure and semantic. Hence, they could be thought as being one a '*clone*' of the other. Thus we apply some algorithms based on Abstract Syntax Trees (AST) from [9], which were originally intended to detect similar pieces of code (clones) on existing programs. Then compatibility for assertions and the usage protocol is carried out by generating ASTs.

All such techniques applied on our assessment procedure help to accomplish a consistent mechanism to assure a fair component integration. As we proceed with our work, reliability is mainly considered, since we base the whole integration process on the challenging Ubiquitous Systems.

The rest of the paper is organized as follows. Section 2 presents our proposal for an integration process. Section 3 illustrates the proposed process with an example. Section 4 discusses implementation alternatives. Conclusions and future work are presented afterwards.

2 Process Framework

We consider a reference model to describe the required engineering practices on our targeted automated Integration Process. Figure 1 depicts partitions showing components through different phases. We add an extra step to distinguish assembly from integration – adapted from [1]. For each phase we distil some steps as is briefly explained below. Numbers imply a non-strict order of execution.

Qualification

1. *Recovery*: Fetch components from a repository or disparate devices.
2. *Assessment*: Evaluate compatibility upon expected behaviours.

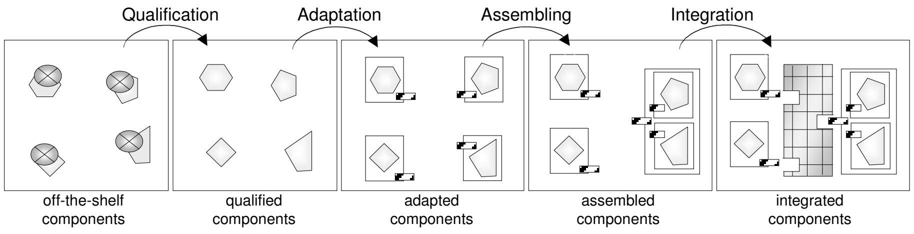


Fig. 1. Component Integration Reference Model

3. *Component Selection*: Analyse options from assessment results and extra requirements (e.g. user preferences).

Adaptation

1. *Adaptation Analysis*: Evaluate conditions from a selected component according to a client component.
2. *Wrapping Selection*: Analyse the most appropriate adaptation strategy.
3. *Tailoring*: Perform the adaptation according to the selected strategy.

Assembly

1. *Assembly Analysis*: Analyse conditions for assembly to create a new application.
2. *Building*: Deployment of additional components for the assembly process.
3. *Composition*: Perform the final application assembly.

Integration

1. *Integration Feasibility*: Analyse conditions to perform connection to an underlying infrastructure (middleware).
2. *Creation*: Construction of additional components for such an integration process.
3. *Glue*: Perform the final application integration.

Next we focus on our Component Assessment procedure at run-time, as part of the Qualification phase. For this, we consider that a component under evaluation should satisfy a certain degree of compatibility with respect to a given requirement specification. We assume such specification describes the required functionality in the form of a component 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 protocol of use for services.

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

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

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

Condidition 2. The protocol of use for services on both components is equivalent. We use $\approx$ to denote “equivalence”[3].

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

[3] Equivalence is measured according to both the element to be compared and the applied technique. From this, different thresholds can be distinguished.

Condition 1 is true when there exists equivalence on corresponding services from both components. A service `sa` of a component `A` is equivalent to a service `sb` of a component `B` when the four next conditions are satisfied:

Condition 1.1. The return type of both services is equivalent[4]:

$$\texttt{typeOf(sa)} \approx \texttt{typeOf(sb)}$$

Condition 1.2. The number of parameters on both services is the same:

$$|\texttt{parms(sa)}| = |\texttt{parms(sb)}|$$

Condition 1.3. Parameter types on both services are equivalent:

$$\forall\ \texttt{pa} \in \texttt{parms(sa)}\ \exists\ \texttt{pb} \in \texttt{parms(sb)}\ /\ \texttt{typeOf(pa)} \approx \texttt{typeOf(pb)}$$

Condition 1.4. Pre and Post-Conditions of sa and sb are similar.

A data type in `sa` (the return type or a parameter type) is equivalent to a data type in `sb` if and only if either both data types are the same or the considered data type in `sa` is `A` and the considered data type in `sb` is `B`:

$$\begin{gathered}\texttt{typeOf(a)} \equiv \texttt{typeOf(b) iif} \\ [\ \texttt{typeOf(a)}=\texttt{typeOf(b)} \vee (\ \texttt{typeOf(a)}=\texttt{A} \wedge \texttt{typeOf(b)}=\texttt{B})]\end{gathered}$$

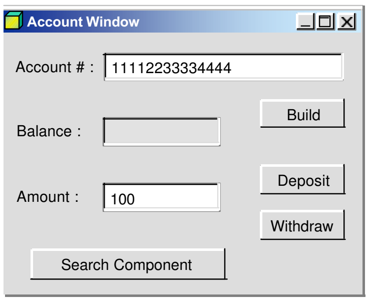
Being `typeOf(a)` a function returning the type of `a` (`a` can be a service or a parameter), and respectively being `A` the component where `a` is included – the same for `b` and `B`.

3 Example

Suppose we have a `window` to operate with banking accounts. A user may create an account by defining the account number and then deposit or withdraw any amount of money. Figure 2 shows the shape of the `window`. The 'build' button allows to create an instance from a typical `BankingAccount` component, whose interface is presented on Figure 3. In some scenario, it is possible that such component is not available to be used by `window`. However, another component with a similar functionality could be recovered from a repository. Suppose the `FinancialAccount` component is found, which presents the interface given below. Then, in order to be sure about their similarity we run the Assessment Procedure as explained in the next section.

```
component FinancialAccount {
        FinancialAccount buildFAccount(string number);
        double obtainBalance();
        void deposit(double quantity);
        void extract(double quantity);
        void transfer(FinancialAccount target, double quantity);}
```

[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.



Account # : 11112233334444

Balance :

Amount : 100

Fig. 2. Account Manager Window

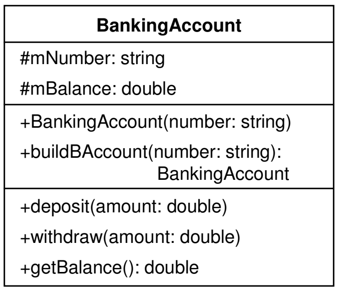

BankingAccount
#mNumber: string
#mBalance: double
+BankingAccount(number: string)
+buildBAccount(number: string): BankingAccount
+deposit(amount: double)
+withdraw(amount: double)
+getBalance(): double

Fig. 3. Banking Account Component

3.1 Assessment Procedure

The Assessment task must verify that Condition 1 and 2 are satisfied – as we pointed out on Section 2. Thus we begin exploring Condition 1.

Interface Equivalence. For Condition 1 to be true we must verify four conditions. However, lets consider first only Conditions 1.1 to 1.3. An equivalence is found on `buildFAccount` and `buildBAccount`. Their number of parameters and parameter types are equal; and their return types are equivalent – both refer to the container component. Next are `getBalance` and `obtainBalance`, since they have no parameters and the same return type. In case of `deposit` and `withdraw` from `BankingAccount` two services from the other component could be similar: `deposit` and `extract`. They match on the return and parameter types, and the number of parameters. `BankingAccount` does not include a service equivalent to `transfer`. However, all of their services are, at a first glance, also offered by `FinancialAccount`.

In order to be accurate on checking Condition 1, Condition 1.4 must be satisfied as well. For this, the comparison of pre and post-conditions from corresponding services is required. For brevity reasons we describe this procedure only for `deposit` (from both components), `withdraw` and `extract`. For such services are specified by using OCL the following assertions.

BankingAccount

deposit
- *pre*: `amount > 0`
- *post*: `mBalance@pre + amount = mBalance`

withdraw
- *pre*: `amount < mBalance@pre`
- *post*: `mBalance = mBalance@pre − amount`

FinancialAccount

deposit
- *pre*: `0 < quantity`
- *post*: `mBalance = mBalance@pre + quantity`

extract
- *pre*: `quantity < mBalance@pre`
- *post*: `mBalance = mBalance@pre − quantity`

SciTePress Science and Technology Publications

To check assertions similarity we derive Abstract Syntax Trees (ASTs). We show on Figure 4 only ASTs for assertions of `deposit` from both components. On each node in the tree we also save a 'type' that is used to operate with its sub-trees. Thus, nodes with values $=$ or $+$ are of type *Interchangeable Operator* (IO), meaning that being `a` and `b` two sub-trees, `a`$+$`b` is the same that `b`$+$`a`. Nodes with value $>$, $<$ or $-$ are of type *Non-Interchangeable Operator* (NIO). Nodes with numbers or variable names are of type *Text* (TXT).

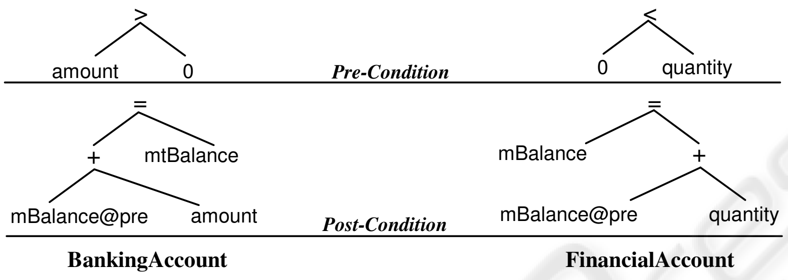


Fig. 4. ASTs for `deposit`'s assertions on both Components

We start evaluating post-conditions from `deposit`. *For this, the root node of both trees are compared and, if they are equal, the respective left and right subtrees are recursively compared.* Being `a` and `b` two trees with IO root nodes, we say that they are equivalent (`a` $\approx$ `b`) iif:

```
(a.leftChild ≈ b.leftChild ∧ a.rightChild ≈ b.rightChild)   ∨
(a.leftChild ≈ b.rightChild ∧ a.rightChild ≈ b.leftChild)
```

In our example, both trees present IO root nodes. Thus, we can compare the left sub-tree of one post-condition with the right sub-tree of the other, and vice versa. This allows to detect the equivalence on both trees and so for the corresponding post-conditions of both services. However, values on leave nodes imply a different case. For this, we consider that two trees with no children with TXT root nodes are equivalent. Hence, all the sub-trees are equivalent regarding both post-conditions, and then also the whole post-conditions are.

ASTs for pre-conditions of both `deposit` services present NIO root nodes, but their values are operators which are indeed *opposite operators*. In fact, the opposite operator to $>$ is $<$, and the transformation can be applied to any of both trees to make the comparison. Hence, also the trees corresponding to pre-conditions are equivalent.

Similar procedures are followed up for each candidate pair of services. In the example, the same operations are made for `extract` and `withdraw`. This process makes clear the real correspondence on `deposit` and `withdraw` from `BankingAccount` with respect to `deposit` and `extract` from `FinancialAccount`.

Since all the assertions are equivalent, Condition 1.4 is fulfilled. Therefore, we can also conclude that Condition 1 is true.

Usage Protocol Equivalence. Now, we must check whether the protocol of use for both components is equivalent. Here it has to be remembered that the usage protocol is described by regular expressions. The usage protocols for `FinancialAccount` component (1), and `BankingAccount` component (2) are given below. Note that we have excluded services `getBalance` and `obtainBalance` to make the example simpler.

$$\text{buildFAccount} \,\textbf{.}\, \text{deposit} \,\textbf{.}\, (\text{extract}+\text{deposit}+\text{transfer})^* \tag{1}$$

$$\text{buildBAccount} \,\textbf{.}\, \text{deposit} \,\textbf{.}\, (\text{withdraw}+\text{deposit})^* \tag{2}$$

For usage protocols, only services actually required, are used for checking. In our example, `transfer` is recognized as an extra service and as such is removed from regular expression (1), as follows:

$$\text{buildFAccount} \,\textbf{.}\, \text{deposit} \,\textbf{.}\, (\text{extract}+\text{deposit})^* \tag{3}$$

Usage protocols comparison is also made deriving ASTs from regular expressions (1) and (3), as can be seen on Figure 5.

ASTs deriving from regular expressions have a different set of operators. The concatenation operator (**.**) is a *Non-Interchangeable Operator* (NIO). Alternative ($+$) is an *Interchangeable Operator* (IO). Repetition ($*$) is an *Unary Operator* (UO) – a tree with just one child. Two nodes corresponding to services with no children in two different trees are equivalent if the services are equivalent from Condition 1 point of view. Thus, as the node labelled with $+$ corresponds to a IO operator, `extract` is equivalent to `withdraw`. Since the trees on Figure 5 are equivalent, Condition 2 is satisfied.

Therefore, as both Condition 1 and 2 are fulfilled, we can infer that `FinancialAccount` offers similar functionalities to those of `BankingAccount`.

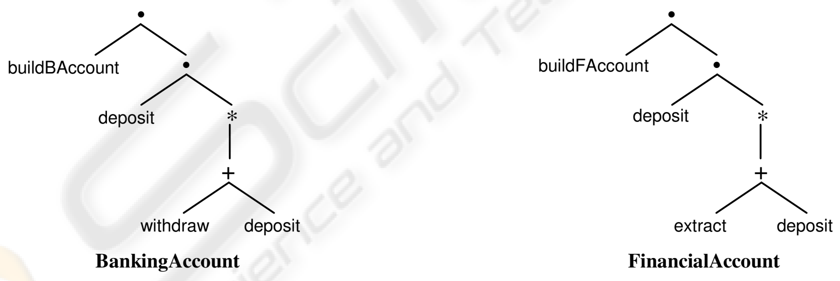


Fig. 5. AST for Usage Protocols

3.2 Searching and Adaptation Procedure

Figure 6 presents a collaboration diagram describing an approach – developed on Microsoft .Net – to achieve the Recovery and Adaptation procedures.

`Window` is an instance of `FormAccount` component – as the one in Figure 2. In order to operate with accounts, it needs to be connected with a component representing

SciTePress Science and Technology Publications

accounts. A special component called `Retriever` searches for components equivalent to a given specification. In our example, `BankingAccount` component represents such a requirement specification provided by `Window`. `Retriever` asks for a component by using a `ComponentManager` which may return a similar component from a `ComponentRepository`. In order to solve likely inconsistencies with the expected component, `ComponentManager` creates an instance from a `Wrapper` component, called `aWrapper`. `FinancialAccount` is found on the `ComponentRepository` and is connected to `aWrapper`. Hence, it is applied the wrapping mechanism named *adapter* [2], to provide the expected signatures form – e.g. `withdraw` and `amount`. A reference of `aWrapper` is returned to `Retriever` by `ComponentManager`. Finally, `Retriever` returns such reference to `window`, so to be able to start operating by being connected to `FinancialAccount`.

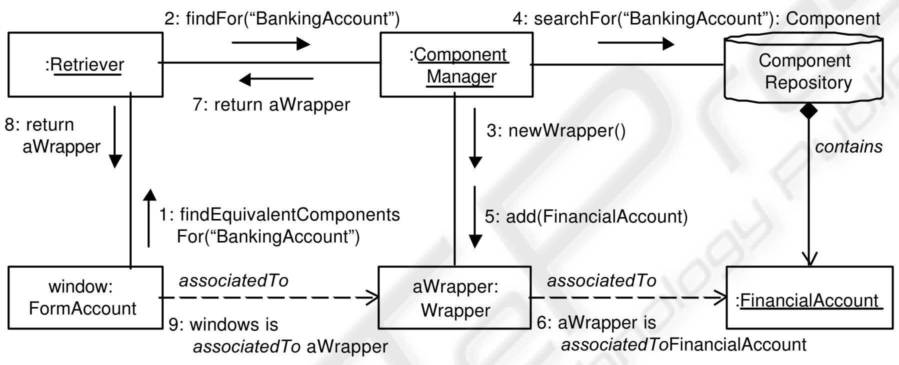


Fig. 6. Recovery and Adaptation procedures

4 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.

4.1 Representing Assertions and Usage Protocol

.NET allows to add meta-data 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. Following is presented both the Constraint class and an example of assertions added to the `deposit` service on `FinancialAccount`.

```
using System;
namespace Components.attribs
{[AttributeUsage(AttributeTargets.Method)]
public class Constraint: System.Attribute
    { protected String mText;
     public Constraint(String text)
       { this.mText=text; }
     ...  }  }
```

```
// Assertions for deposit
   [Precondition(‘‘<(0,amount)")]
   [Postcondition(‘‘=(mBalance,
          +(mBalance@pre,amount))")]

   public void deposit(double
                             amount)
                { mBalance+=amount;}
```

Regular expressions for the usage protocol are represented in a similar way, where the ambit in this case 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.

4.2 Recovering the Interface

In order to inspect the set of members of any element, .NET includes the *Reflection* mechanism. This can be used to recover the set of methods from components to be evaluated. Reflection can be of substantial help in cases where components are discovered from some mobile devices.

5 Conclusions

We have presented a preliminary scheme to address the automation of a Component-base Integration Process for Ubiquitous Systems. We are particularly working on the Qualification and Adaptation phases. Our approach of component Assessment is based on meta-data added to components, describing assertions and the usage protocol by means of OCL.

We have developed a simple prototype on Microsoft .Net to implement our approach. As reliability is our main concern, selecting appropriate methods, techniques and languages, must be accurately accomplished. This is the emphasis of our next development in this area.

Acknowledgments

We would like to thank Dr. Juan Carlos Augusto, from School of Computing and Mathematics, University of Ulster, Newtownabbey, UK. His participation provides a meaningful complementary view giving accuracy and confidence to the project.

References

1. Brown, A., Wallnau, K.: Engineering of Component-Based Systems. In: IEEE 2^{nd} ICECCS'96, Montreal, Canada (1996) 414–422
2. Flores, A., Polo, M.: Dynamic Assembly & Integration on Component-based Systems. In: 4^{th} JIISIC, Madrid, España (2004) 349–360
3. Flores, A., Cechich, A.: Quality Considerations on Ubiquitous Systems. In: II Workshop de Ingeniería de Software, at JCC'02, Copiapo, Chile (2002)

4. Garlan, D.e.: Software Architecture-based Adaptation for Pervasive Systems. In: ARCS'02. Volume 2299 of LNCS., Karlsruhe, Germany (2002) 67–82
5. Flores, A., Polo, M.: Considerations upon Interoperability on Pervasive Computing Environments. In: 6th WICC, pp. 162-166, Neuquen, Argentina (2004)
6. Flores, A., Augusto, J.C., Polo, M., Varea, M.: Towards Context-aware Testing for Semantic Interoperability on PvC Environments. In: IEEE 17th SMC'04, special session: CRIPUC, The Hague, Netherlands (2004) 1136–1141
7. Kirani, S.: Specification and Verification of Object-Oriented Programs. PhD thesis, Computer Science, University of Minnesota, Minneapolis, USA (1994)
8. Brada, P.: Towards Automated Component Compatibility Assessment. In: 6th Wrkshp on Comp-oriented Prog, at ECOOP'01, Budapest, Hungary (2001)
9. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: ICSM'98, pp. 368-377, Maryland, USA (1998)

SciTePress
Science and Technology Publications