Materializing Architecture Recovered from OO Source Code in Component-Based Languages

Zakarea Al-Shara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony, Anas Shatnawi

Materializing Architecture Recovered from OO Source Code in Component-Based Languages

Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine,
Hinde Lilia Bouziane, Christophe Dony, and Anas Shatnawi

UMR CNRS 5506, LIRMM, University of Montpellier, France
{alshara,seriai,tibermacin,bouziane,dony,shatnawi}@lirmm.fr

Abstract. In the literature of software engineering, many approaches have been proposed for the recovery of software architectures. These approaches propose to group classes into highly-cohesive and loosely-coupled clusters considered as architectural components. The recovered architecture plays mainly a documentation role, as high-level design views that enhance software understandability. In addition, architecture recovery can be considered as an intermediate step for migration to component-based platforms. This migration allows to fully benefit from all advantages brought by software component concept. For that, the recovered clusters should not be considered as simple packaging and deployment units. They should be treated as real components: true structural and behavior units that are instantiable from component descriptors and connected together to materialize the architecture of the software. In this paper, we propose an approach for revealing component descriptors, component instances and component-based architecture to materialize the recovered architecture of an object-oriented software in component-based languages. We applied our solution onto two well known component-based languages, OSGi and SOFA.

1 Introduction

Component Based Software Development (CBSD) has been recognized as a competitive principle methodology for developing modular software systems [4]. It enforces the dependencies between components to be explicit through provided and required interfaces. Moreover, it provides coarse grained high-level architecture views for component-based (CB) applications. These views facilitate the communication between software architects and programmers during development, maintenance and evolution phases [11].

Otherwise, object-oriented (OO) have fine-grained entities with complex and numerous implicit dependencies [7]. Usually, they do not have explicit architectures or even have "drifted" ones. These adversely affect the software comprehension and makes these software systems hard to maintain and reuse [6]. Thus migrating OO software to CB one should contribute to gain the benefits of CBSD [9].

The process of migrating OO applications to CB ones involves two major steps: architecture recovery and code transformation [13]. The first step consists

of identifying reusable components from legacy OO systems. A component is represented by a cluster of classes where its provided and required interfaces are represented by a set of provided and required methods respectively. The main challenge of this step is to find the best clusters compared to the component definitions which reflect the right software architecture. The second step aims at creating programming level components by transforming and generating a component code based on the OO one. The main problem of this step is to obtain a code which conforms to component principles: encapsulation, interface-based interaction, component instantiation, etc. [25].

Architecture recovery has been largely treated in the literature. Many approaches have been proposed to recover software architectures from legacy OO source code [17, 5, 18, 16, 3]. In contrast, only few approaches have been proposed for really transforming OO code into CB one [24, 7, 14]. In addition, these approaches have only partially address the code transformation step (c.f. Sec 6).

In this paper, we propose an approach for transforming OO code to CB one guided by the recovered architecture of the corresponding OO software. This approach allows to reveal component descriptors, component instances and component-based architecture to materialize the recovered architecture. To validate this approach, we applied it to transform Java code to two well known component-based languages; OSGi [23] and SOFA [19].

The remainder of this paper is organized as follows. Section II discusses the problem statement. Section III presents the transformation of OO code to CB one. Section IV presents how the proposed solution is mapped onto OSGi and SOFA. Section V presents the discussion about our solution. Section V discusses related work. Finally, Section VI contains some concluding remarks and gives directions to future work.

2 Problem Statement

To better illustrate our approach aiming to transform OO code to CB one, first, we introduce in this section an example of a simple Java application. Second, we present the expected architecture recovered by analyzing this application. Finally, we illustrate the problem of OO code transformation guided by this architecture.

2.1 Running Example

Fig. 1 shows an example of a simple Java application that simulates the behavior of an information screen (e.g. a software system which displays on a bus's screen information about stations, time, etc.). *ContentProvider* class implements methods which send text messages (instances of *Message*), and time information obtained through *Clock* instances based on the data returned by *TimeZone* instances. The *DisplayManager* is responsible for viewing the provided information through a *Screen*.

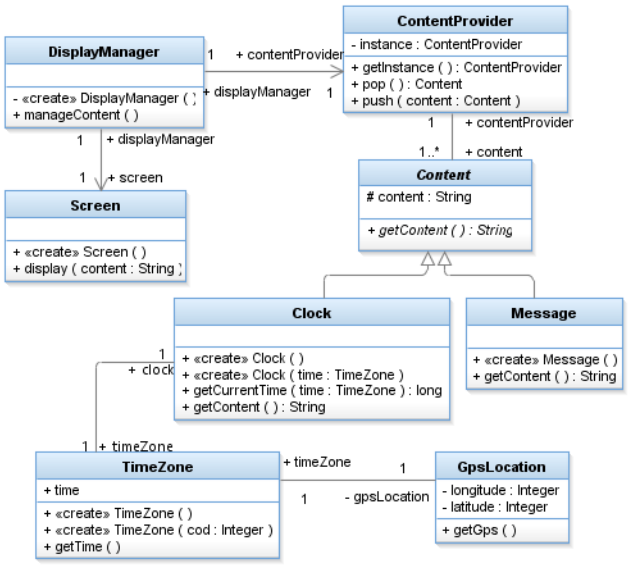

Fig. 1: *Information-screen* class diagram

2.2 Component-based Architecture Recovery

Architecture recovery approaches consider a component as a cluster of classes [17, 5, 18, 16, 3]. In our previous work [18, 16], we have proposed an approach which aims to recover component-based architectures from OO source code. Fig.2 shows the object-component mapping model used in this approach. In this model a cluster is composed of two types of classes: internal classes and boundary classes. The internal classes are those that do not have dependencies with other classes placed into other clusters. In contrast, the boundary classes are those that have dependencies with classes placed into other clusters.

Fig. 3 shows the result of architecture recovery step applied on our example. The recovery step identifies four clusters (components), where each cluster may contain one or several classes. We consider a component-based architecture as a set of components connected via interfaces, where interfaces are identified from boundary classes. For example, the component *DisplayedInformation* connected to *ContentProvider* component through two interfaces. The first interface declares *getCurrentTime* method which is placed in class *Clock* and *getContent* method from class *Clock*. The second one declares *getContent* method from class *Message*.

2.3 Code Transformation: Component Source Code Generation Based on OO Source Code

Clusters of classes identified from architecture recovery represent the primary implementation code of components. This code should be transformed to match

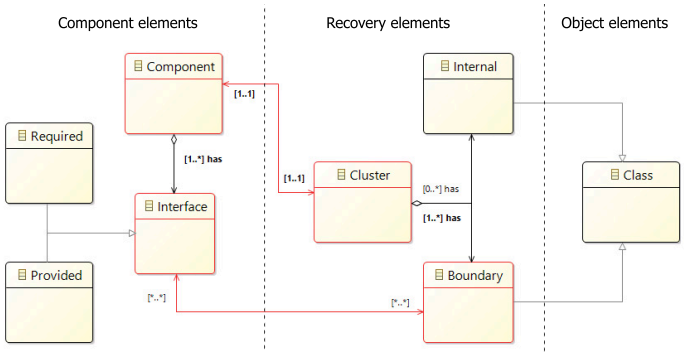

Fig. 2: Object-to-component mapping model

targeted CB languages. These languages can be classified into two main categories. The first category distinguish the language used for describing components and architectures (architecture description language) from the language used to implement components (programing language) like SOFA [19]. The second category use the same language for describing architecture descriptions and component implementations like COMPO [12]. In our work we focus on transforming OO code to one written using CB language of the first category. This transformation allows to reuse classes of recovered clusters as the implementation of the target components. Table 1 summarizes the main structural elements of languages of this category. These consist of:

1. Structural elements that define component descriptions:
   (a) Component interfaces: the component descriptor need to defined provided and required interfaces. all interactions between components must be done through these interface.
   (b) Implementation reference: the component descriptor need to defined references of its component implementation source code.
   (c) Component instantiation: the component descriptor need to defined how its component can be instantiated.
2. Architecture description: it describes the structure of component-based systems in terms of component instances and component assembly. It ignore components implementation details and interactions.

Our approach aims at generating structural elements composing component descriptors and architecture description starting from source code of recovered clusters. In our previous work [20], we have proposed an approach that transform dependencies between clusters to be interface-based ones. This approach presented component interfaces structural elements. In this paper we complete the transformation by addressing the remains structural elements of component descriptors; implementation references and component instantiation. In addition to reveal the CB architecture.

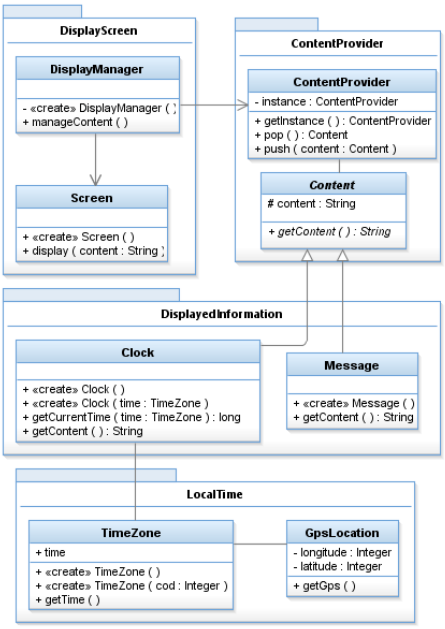

Fig. 3: *Information-screen* architecture recovery

3 Transforming OO Code to CB One

3.1 Generating Component Descriptor and Reference of its Implementation

Our approach uses the concept of class used in OO to express component descriptors. Hence, a class will represent the component descriptor. For example, the descriptor of *DisplayedInformation* component translated by creating a new class *DisplayedInformation*. Where the component descriptor has describe their interfaces, the same concept of interface in OO languages is used to describe component interfaces. Then each provided interface has an OO interface that explicit its services (method signatures). The component descriptor must has the reference of its implementation of all provided interface services. For example, Listing 1.1 shows how the provided interfaces for component *DisplayedInformation* are created. But, what if two interfaces have the same method signature? the descriptor can not implement two services in the same descriptor (this is the case in Java, but in C++ and C# we can implement the same services that have the same signature by referencing the interface name before the implemented methods). For example, component *DisplayedInformation* provides two interfaces and the two interfaces have a method with the same signature(*getContent()*). Consequently, we should provide each interface by a component port.

Table 1: Object-based Component Model Specifications [8]

Component Models	Language of implementation	Interfacs type	Component Descriptor	Component instance
EJB [2]	Java	Operation-based	Yes	Single Object
Fractal citefractal	Java, C#, .Net	Operation-based	Yes	Single Object
JavaBeans [21]	Java	Operation-based	Yes	Single Object
COM [1]	OO languages	Operation-based	Yes	Single Object
OpenCOM [10]	OO languages	Operation-based	Yes	Single Object
OSGi [23]	Java	Operation-based	No	Many Objects
SOFA 2.0 [19]	Java	Operation-based	Yes	Single Object
CCM [22]	Language independent with OO implementation	Operation-based & Port-based	Yes	Single Object
COMPO [12]	COMPO	Operation-based & Port-based	Yes	Single Object
Palladio [15]	Java	Operation-based	Yes	Single Object
PECOS [26]	OO languages	Port-based	Yes	Single Object

Listing 1.1: Provided interfaces for *DisplayedInformation* component

```
public interface ITime {
  public String getContent();
  public long getCurrentTime(ITimeZone timeZone);
}
public interface IMessage {
  public String getContent();
}
```

The explicit services provided by a component interface are associated with a port. In our approach, we use the inner-class concept used in OO to represent component ports. Thus, each port is described by an inner-class associated with its interface. For example, in Listing 1.2, the *PortTime* inner-class is created to implement *ITime* interface provided by component *DisplayedInformation*, as same as *PortMessage* inner-class. Moreover, the references of each inner-class (port) are provided by its component (e.g. *portTime* and *portMessage* class-variables) for binding components.

Listing 1.2: Descriptor and ports for *DisplayedInformation* component

```
public class DisplayedInformation{
  public static ITime portTime;
  public static IMessage portMessage;
  private class PortTime implements ITime{
      @Override
      public String getContent() {//TODO: add behaviore implementation }
        @Override
      public long getCurrentTime(ITimeZone timeZone) {//TODO: add behaviore implementation }
  }
  private class PortMessage implements IMessage{
      @Override
      public String getContent() {//TODO: add behaviore implementation }
  }
}
```

3.2 Component Instantiation

Mapping object instances to component instances In OO, an instance consists of state and behavior, the state is stored in variables and exposes its behavior through methods. Object hides its internal state where methods operate in an object internal state to provide services through object-to-object communication (encapsulation). However, the recovered component is viewed as a set

of one or more cooperating classes. Thus, we infer component instances from a set of class instances belonging to the same component, where the component state is the aggregated state of these instances, and the component behavior is published through the component interfaces. For example, in Fig. 4, we have three object call graphs for a component consisting of five classes (*A, B, C, D, E*). We can observe that:
(1) The component instance has three different releases (Fig. 4 (a), (b) and (c)).
(2) The component instance could have many class instances of the same type. For example, Fig. 4-(c) have two class instances from type *E* (e1 and e2).
(3) The client needs to have references to the class instances that provide services/methods for them. For example, the classes that implement the provided component services are *A* and *B*. Then, the client needs to reference instances of type *A* and *B* to get their required services. After that, instances of type *A* and *B* are responsible to communicate with other instances to complete its services. And therefore, the classes that have the component provided services are considered as the only entrance to component instance.

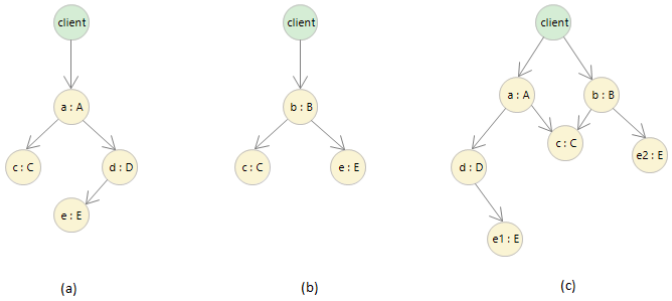

Fig. 4: Different release of the same component instance

Based on our interpretation of the component instance, the set of class instances that constitute a component instance should behave as a single unit. Then, we need to update component descriptor to manage the links between class instances that form a component instance. We propose to delegate provided interface methods in the component descriptors to real ones. For example, Listing1.3 describes the update of the descriptor of *DisplayedInformation* component. The descriptor has references of the classes types that are responsible to provide component services *Clock* and *Message*. After that, the delegations of provided services is done through component ports by using the real class instances that have these services. It is worth noting that we used the lazy instantiation of these class instances (delaying the instantiation of class instance until the first time it is needed) for performance reasons.

Listing 1.3: Component descriptor with its behaviors

```
public class DisplayedInformation{
  protected static ITime portTime;
  protected static IMessage portMessage;
  //Boundary Classes
  Clock clock = null;
  Message message = null;
  public DisplayedInformation() {
    //initializing component ports
    portTime = new PortTime();
    portMessage = new PortMessage();
  }
  private class PortTime implements ITime{
    @Override
    public String getContent() {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getContent();
    }
    @Override
    public long getCurrentTime(ITimeZone timeZone) {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getCurrentTime(timeZone);
    }
  }
  private class PortMessage implements IMessage{
    @Override
    public String getContent() { //lazy instantiation
      if(message == null){
        message = new Message();}
      return message.getContent();
    }
  }
}
```

Creating Component Instances The services of a component can not be used directly, the component descriptor must first be instantiated. Like in OO programs, we need a constructor to create a component instance and initialize its state. The constructor of the component should be placed into the component descriptor. In addition, the descriptor implements the component services through component interfaces using associated ports. Thus, we create a default constructor (constructor without parameters) that initializes component ports. Listing 1.4 describes the default constructor of component *DisplayedInformation* and how it creates its ports (*PortTime* and *PortMessage*).

Initializing component state depends on the constructors placed into classes that have provided methods to other components (e.g. *Clock* and *Message* into *DisplayedInformation* component). For example, class *Clock* has two constructors, the first one without parameters (default constructor) and the second one with a single parameter of type *ITimeZone*. So two possible ways to create an instance of type *Clock*. Therefore, the component descriptor should provide all possible ways to initialize its instances. Consequently, *initialize* methods are created with different parameters to apply component configurations. For example, *DisplayedInformation* component have two classes that can be accessed from outside components (*Clock* and *Message*), and each of them has default con-

structor while *Clock* class has one more with *ITimeZone* parameter. Therefore, *initialize* methods are created and has *ITimeZone* parameter (see Listing 1.4).

Listing 1.4: Component constructors and initializers

```
public class DisplayedInformation{
...
  public DisplayedInformation() {
    //initializing component ports
    portTime = new PortTime();
    portMessage = new PortMessage();
  }
  public initialize(ITimeZone timeZone) {
    clock.setTimeZone(timeZone);
  }
}
```

Now, we can simply create an instance of the component using its constructor using OO instantiation and then initialize the instance using appropriate initializer. For example, an instance of *DisplayedInformation* component is created by its constructor using *new* keyword. Listing 1.5 differentiates the refactoring resulted from our approach (*ComponentClient*) and the original source code (*ClassClient*).

Listing 1.5: Component instantiation

```
public class ClassClient{
  Clock clock = new Clock(timeZone);
  clock.getCurrentTime();
}
public class ComponentClient{
  DisplayedInformation info = new DisplayedInformation();
  info.initialize(timeZone);
  info.portTime.getCurrentTime();
}
```

3.3 Reveal Component-based Architecture

An architecture description describes the structure of component-based systems in terms of component instances and binding. Therefore, to reveal a CB architecture, we need to identify its component instances and the binding between these instances. We can identify the component instances by analyzing the instantiation statements of its implementation. We can identify the binding between these instances based on the invocation of its services.

Identify component instances We first statically analyze the source code to check whether to create a new component instance or to use an existing one. The analysis is based on statement scope (i.e. in the same code block) and obliterates state (i.e. the second instantiation statement obliterates the state of the instance resulted from first one). The previous component instance can be replaced by a set of its class instances if these set at the same scope and no one obliterates the state of another one. For example, in Listing1.6, the IF-BLOCK into class

ClassClient instantiates an object of type *Clock* and another of type *Message*. However, the proposed approach replaces the two instances with a component instance of type *DisplayedInformation* (*info1*) because they are in the same scope and each one does not obliterate the state of another. An example of the scope condition is obviously shown by defining *info1* and *info2*, where each of them belongs into different scopes. Defining *info2* and *info3* provides an example of obliteration state condition, where *message2* will obliterate the state of *message1* if it translated to one component instance. Listing1.7 shows the component instances that have been identified from Listing1.6.

Listing 1.6: Refactoring instantiation from OO code into CB one

```
public class ClassClient{
  if(condition)
  {
    Clock clock = new Clock(timeZone);
    Message message = new Message();
  }else{
    Message message1 = new Message();
    ...
    Message message2 = new Message();
  }
}
public class ComponentClient{
if(condition)
  {
    DisplayedInformation info1 = new DisplayedInformation();
  }else{
    DisplayedInformation info2 = new DisplayedInformation();
    ...
    DisplayedInformation info3 = new DisplayedInformation();
  }
}
```

Listing 1.7: Identified CB instances for architecture discriptor

```
//Darwin ADL
inst
info1 : new DisplayedInformation();
info2 : new DisplayedInformation();
info3 : new DisplayedInformation();
```

Identify component binding Binding between component instances is used to establish interactions between these instances. An instance of component binds to another one to provide or required services through their interfaces. Therefore, we can identify the bindings between components based on service invocations between them where components must firstly bind to provide or required services. For example, in Listing1.8, *ContentProvider* invokes a service *getCurrentTime* from *DisplayedInformation*, so the binding between these two component must be established before. Therefor, we can statically analyze these invocations between components to identify bindings (see Listing 1.9). Fig 5 shows the architecture recovery (c.f. Sec. 2.2) and a snapshot of architect description for our running example. The architect description describes component instances

and its binding between *DisplayedInformation* and *ContentProvider* component instances.

Listing 1.8: Refactoring instantiation from OO code into CB one

```
public class ContentProvider{
  public void push(DisplayedInformation info1){
    String time = info1.portTime.getCurrentTime();
    }
}
```

Listing 1.9: Refactoring instantiation from OO code into CB one

```
//Darwin ADL
inst
content : new ContentProvider();
information : new DisplayedInformation();
bind
content.I1 -- information.ITime
}
```

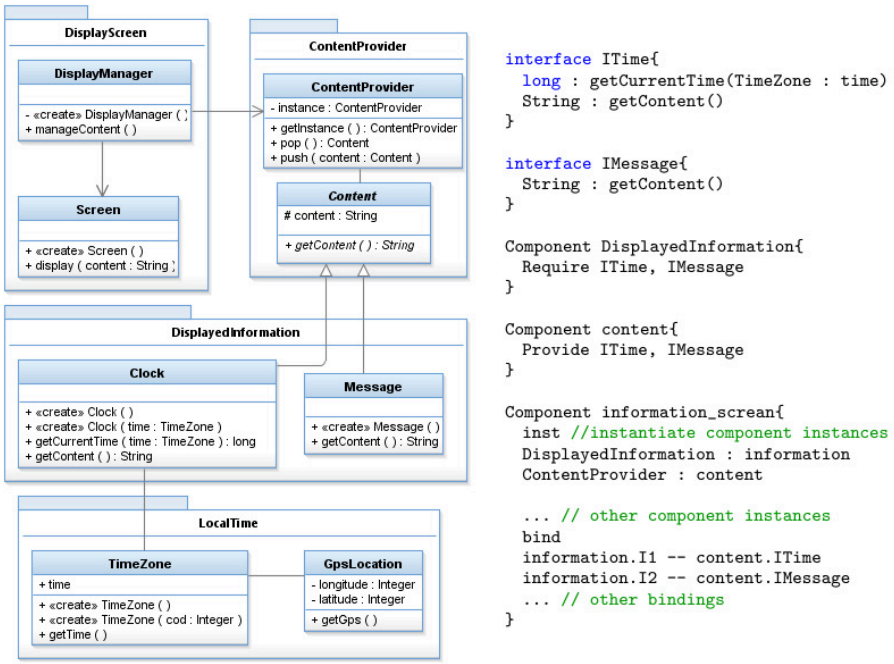

Fig. 5: *Information-screen* architecture recovery and Darwin ADL for *DisplayedInformation* and *ContentProvider*

4 Mapping the Proposed Solution onto Component Models

In this section we describe how our proposed solution is easily mapped onto existing component models. We have chosen two well known component models, OSGi and SOFA, to explain the ease of the mapping.

4.1 Mapping from Java to OSGi

OSGi is a set of specifications that define a component model for a set of Java classes [23]. It enables component encapsulation by hiding their implementations from other components by using services. The services are defined by standard Java classes and interfaces that are registered into a *service registry*. A component (bundle) can register and use services through the *service registry*.

Listing 1.10: *DisplayedInformation* component describtor and its interface

```
public class DisplayedInformation implements IDisplayedInformation{ /* Contents... */ }
public interface IDisplayedInformation {
  public InterTime portTime = DisplayedInformation.portTime;
  public IMessage portMessage = DisplayedInformation.portMessage;
}
```

To map our transformed code onto OSGi framework, we firstly create an interface (Java interface) to represent the contract of provided component instance. For example, Listing 1.10 shows how we created an interface for *DisplayedInformation* component. Hence we suggest that a component binds through its port associated with a provided interface, then both ports *InterTime* and *IMessage* must be accessed by other components. After that, a metadata for both provided component *DisplayedInformation* and required component *ContentProvider* must be specified. The metadata specified through XML files using the declarative services model. For example, Listing 1.11 describes how *DisplayedInformation* component provides its instances as object interfaces with type *IDisplayedInformation*. And Listing 1.12 describes how *ContentProvider* component uses the provided instances. When both components are activated at runtime, the binding is established between them. Listing 1.13 describes how *ContentProvider* gets an instance of *DisplayedInformation* and call its method *getContent()* through port *portMessage*.

Listing 1.11: DisplayedInformation.xml file to provide the instances of *DisplayedInformation*

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="DisplayedInformation">
   <implementation class="DisplayedInformation"/>
   <service>
      <provide interface="IDisplayedInformation"/>
   </service>
</scr:component>
```

Listing 1.12: ContentProvider.xml to bind the instances of *DisplayedInformation*

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="ContentProvider">
   <implementation class="ContentProvider"/>
   <reference bind="setDisplayedInformation" cardinality="1..n"
        interface="IDisplayedInformation" name="DisplayedInformation" policy="static"
        unbind="setDisplayedInformation"/>
</scr:component>
```

Listing 1.13: binding between *DisplayedInformation* and *ContentProvider*

```
public class ContentProvider implements Inter_ContentProvider{
   public synchronized void setDisplayedInformation(IDisplayedInformation information) {
    information.portMessage.getContent();
   }
}
```

4.2 Mapping from Java to SOFA 2.0

SOFA is a platform for software components that uses a component model with hierarchically nested components (composite components). It describes a component by its frame (component descriptor) and component architecture. The frame is a black-box view of the component that defines its provided and required interfaces. It provides a metadata (XML files) to describe provided and required services. Components are interconnected via bindings among interfaces using connectors.

Listing 1.14: DisplayedInformation.xml to provide the instances of *DisplayedInformation*

```
<?xml version="1.0"?>
<frame name="DisplayedInformation">
  <provides name="DisplayedInformation" itf-type="sofatype://IDisplayedInformation"/>
</frame>
```

Listing 1.15: ContentProvider.xml to bind the instances of *DisplayedInformation*

```
<?xml version="1.0"?>
<frame name="ContentProvider">
  <requires name="DisplayedInformation" itf-type="sofatype://IDisplayedInformation"/>
</frame>
```

Listing 1.16: binding between *DisplayedInformation* and *ContentProvider*

```
public class ContentProvider implements SOFALifecycle, Runnable, SOFAClient {
  IDisplayedInformation info = null;
  // Called during initialization of the component.
  public void setRequired(String name, Object iface) {
    if (name.equals("DisplayedInformation")) {
      if (iface instanceof IDisplayedInformation) {
      //get DisplayedInformation instance
        info = (IDisplayedInformation) iface;
        info.portMessage.getContent();
      }}}
}
```

5 Discussion

We can deploy recovered cluster of classes directly onto existing component models without using our approach. Indeed, we can transform each class into a component and then assemble these components that belong to the same cluster using component composition property as a composite component. However, to compare our approach with the composite component approach, we need first to study the component composition types and component models that support these types. Table 2 shows the selected object-based component models and composition supported composition types. Two types of component compositions; the first one is horizontal composition, and the second type is vertical composition. The horizontal composition means that components can be binded through their interfaces to construct component applications. The second type, vertical composition, describes the mechanism of constructing a new component from two or more other components. The new component is then called composite because they are themselves made of more elementary components called internal components. Internal components could be accessible or visible from clients (delegation) or not (aggregation).

We can observe from Table 2 that there are five component models that did not support vertical composition at all (*EJB*, *JavaBeans*, *OSGi*, *CCM* and *Palladio*). Four of them provide vertical aggregation composition and six models support vertical delegation composition. However, vertical delegation composition is not appropriate because clients can access or view the internal components (violates component encapsulation). Consequently, the vertical aggregation composition could be replaced by our approach, but there are just two component models that support it.

Table 2: Composition type in object-based component models

Component Models	EJB	Fractal	JavaBeans	COM	OpenCOM	OSGi	SOFA 2.0	CCM	COMPO	Palladio	PECOS
Vertical Composition	No	Yes	No	Yes	Yes	No	Yes	No	Yes	No	Yes
Aggregation		X		X	X				X		
Delegation		X		X	X		X		X		X

6 Related Work

Transforming OO applications to CB ones has two types of related works. The first relates to CB architecture recovery, and the second relates to code transformation from OO applications to Component-oriented ones. Many works have proposed for recovering CB architectures from OO legacy code. A survey on these works is presented in [17] and [5]. However, only few works have beenproposed a transformation from OO code to CB one.

The approach proposed by [14] applies in transforming Java applications to OSGi. The approach uses OO concepts and patterns to wrap cluster of class to components. However, they did not deal with component instantiation, where they still used instances in terms of OO. Another approach for transforming Java

applications into the JavaBeans framework is proposed in [7]. They developed an approach that can generate components from OO programs using a class relations graph. This method did not deal with a component as a set of classes, the authors assume that each class is transformed to a component. Therefore, it can not treat the cluster of classes recovered from architecture recovery methods.

One of the closest works to our approach is proposed by [24]. They used dynamic analysis to define component interfaces and component instances. The idea of their work consists of four steps. The first one is an extraction of object call graphs. The second step is transforming the object call graph into a component call graph. The third step identifies component interfaces based on the connections between component instances. The last step deals with component constructors and its parameters. In contrast to our work, they use dynamic analysis and execution trace, where they supposed the use cases of the recovered applications exist and fully cover all execution cases. Moreover, they suppose that two component instances may have intersected states, where a class instance can be shared between two components which violate the principle of component encapsulation.

7 Conclusion

In this paper, we proposed an approach to transform recovered components from object-oriented applications to be easily mapped to component-based models. We refactored clusters of classes (recovered component) to behave as a single unit of behavior to enable component instantiation. Our approach guarantees component-based principles by resolving component encapsulation and component composition using component instances. The encapsulation of components is guaranteed by transforming the OO dependencies between recovered components which was proposed in our previous work [20]. Moreover, both principles applied by refactoring a recovered component source code to be instantiable, where the provided services are consumed by the component instance through its interfaces (component binding). We have shown that the source code resulted from our approach can be easily projected onto object-based component models. We illustrated the mapping onto two well known component models, OSGi and SOFA. The illustration results shows that our approach facilitates the transformation process from OO applications into CB ones. Moreover, it effectively reduces the gap between recovered component architectures and its implementation source code.

References

1. D. Box. Essential com. object technology series, 1997.
2. Oracle E.E. Group. Jsr 220: Enterprise javabeanstm,version 3.0 ejb core contracts and requirements version 3.0, final release, May 2006.
3. A. Shatnawi et al. *Mining Software Components from Object-Oriented APIs, ICSR 2015. Proceedings.* Springer International Publishing, 2014.

4. A.Bertolino et al. An architecture-centric approach for producing quality systems. *QoSA/SOQUA*, 3712:21–37, 2005.
5. D. Birkmeier et al. On component identification approaches classification, state of the art, and comparison. In *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2009.
6. E. Constantinou et al. Extracting reusable components: A semi-automated approach for complex structures. *Information Processing Letters*, 2015.
7. H. Washizaki et al. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 2005.
8. I. Crnkovic et al. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 2011.
9. K. Lau et al. Software component models. *Software Engineering, IEEE Transactions on*, 2007.
10. M. Clarke et al. An efficient component model for the construction of adaptive middleware. Springer Berlin Heidelberg, 2001.
11. M. Shaw et al. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
12. P. Spacek et al. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14. ACM, 2014.
13. R. Kazman et al. Requirements for integrating software architecture and reengineering models: Corum ii. In *Reverse Engineering, 1998. Proceedings.*, 1998.
14. S. Allier et al. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Software Architecture (WICSA)*, 2011.
15. S. Becker et al. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07. ACM, 2007.
16. S. Chardigny et al. Extraction of component-based architecture from object-oriented systems. In *Software Architecture, 2008. WICSA 2008.*, 2008.
17. S. Ducasse et al. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 2009.
18. S. Kebir et al. Quality-centric approach for software component identification from object-oriented code. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012.
19. T. Bures et al. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications.*, 2006.
20. Z. Alshara et al. Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015. ACM, 2015.
21. Sun Microsystems. Javabeans specification, 1997.
22. OMG. Omg corba component model v4.0, 2011.
23. Osgi Service Platform. The osgi alliance, release 6, 2015.
24. A. Seriai. Enactment of components extracted from an object-oriented application. In *Software Architecture*. Springer International Publishing, 2014.
25. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.
26. M. Winter. The pecos software process. In *Workshop on Components-based Software Development Processes, ICSR*, 2002.