Whole-Part based Composition Approach: a Case Study

Nicolas Belloir, Fabien Romeo and Jean-Michel Bruel
LIUPPA – Université de Pau et des Pays de l'Adour
F-64000 Pau, France
contact: belloir@univ-pau.fr

Abstract

Component Based Software Engineering is a good response to actual systems needs: flexibility and adaptability. In this context, design techniques based on software composition and verification techniques proving that the design properties are well implemented, are current and actively studied problematics. In this paper we present a composition design technique based on a formally defined theory: the whole-part relationship. We show, with an in-house verification framework, how to verify an implementation of composition properties. We illustrate our approach with a case study.

1. Introduction

Our research group is dealing more specifically with one of the limitations of the current approaches supporting component-based software engineering (CBSE): the fact that software composition is mainly treated at assembly-time only (running components). This limitation is particularly sensible for adaptability concerns, which is difficult to achieve when dependencies between components are not formally specified. These adaptations, and hence the concerns focussing on dependencies, are necessary for a correct components integration. One must determine the properties of the assemblies through component compatibility at deployment time in particular [16]. To summarize, as it has been pointed out by numbers of reports [1, 15] or recent CBSE workshops [5], components are not enough conceived to take into account their composability, reusability, and hence flexibility. Our focus is then not on the components themselves, but on their composition, and more specifically, on their composition expressed as soon as possible in the development process. We are using a Platform Independent Model (PIM) approach following the current directions promoted by the OMG [10]. We then consider components at the modeling level. Our modeling approach is based on the recently adopted UML 2.0 notation [11], which has made significant progress in component support [18]. Unfortunately improvements have only been made on the notation itself [6].

We are working on an approach [3] based on an overall theory: the whole-part relationship (WPR). This theoretical framework is used to specify the software composition at design level and to constraint the component-based implementations. It is based on the formal definition of the properties of this relationship. The composition can then be specified by selecting one or more of these properties. In this article we will describe how to concretely use such a framework and we will detail the associated implementation environment we have developed. This environment allows us to easily implement compositions by a dedicated support of the possible properties we have defined. It is also possible to check, especially at run-time, the assembly properties of the components. We will illustrate its use through a case study consisting of a *CoffeeMachine*, composed of two subcomponents, a *Coiner* and a *DrinkMaker*.

The paper is organized as follows: in section 2 we will make a short recall of our proposal, in section 3 we will describe our implementation environment, in section 4 we will illustrate its use by a case study and we will conclude in section 5 about the benefits and the ongoing developments of our approach.

2. Overall description of the approach

Our approach aims at improving the development of assemblies of components. Our composition approach is based on a reflexive compositional hierarchy: the WPR. In this approach, any composition is seen as a high-level component (Whole) composed of subcomponents (Parts). This Whole component is based on the services of its Part components, and itself provides more elaborated services. This idea is close to those adopted by the recent WCOP workshop [5] in which it is for example concluded: ”*In fact, one can think of a component-based system as a triangle. At the top node a component-based system consists only of one component that actually represents the whole architecture of the corresponding application. A zoom operation can then be used to decompose the application* ”. This approach has also some similarities with the *Composite* design pattern [9]. It is different, however, in the sense that we are more interested in the formalization of the properties of the composition relationship than in applying a pattern to a defined design. Some other approaches aim at the formal definition of composition, such as, for example, those using the B language [12]. Nevertheless these approaches often require from the designers some knowledge and skills in the use of formal notations. Our goal is to provide well defined properties, hiding in a sense the complexity of the formal definition.

2.1. The Whole-Part Relationship

Our theoretical framework is based on an adaptation to CBSE of a formalization of the WPR [2]. We have extended the semantic properties of the WPR by adapting its formal base to software composition. We have determined which properties to apply to component composition and defined formally these properties. We ground our approach on metamodeling and assertions in order to constrain the specification of composition relationships. Constraints are added to components at implementation time via generated contracts [3].

Dependency variations between a Whole and its Parts may be expressed through the choice of a precise set of basic characteristics. Some of them are always present in a WPR and some are optional. We then split these properties into two categories: primary properties (properties that a WPR must always respect) and secondary properties (properties that specialize a WPR in specific subtypes). The primary properties are the properties which a relation must possess in order to be qualified as a WPR: binary nature of the relation, asymmetry at component level, anti-symmetry at instances level, existence of at least an emergent property and of a resulting property. The secondary properties are the properties which characterize the type of WPR: encapsulation, lifetime dependency, transitivity, shareability, separability, mutability, and existential dependency.

Applied in the context of CBSE, some WPR primary properties can be regarded as heuristics only (this is the case for the resulting and emergent properties). The precise selection of secondary properties of the composition link between two components helps for example to apprehend the definition of the behavior of the assembly of components. This is an important feature for the community [5].

2.2. Composition properties

In this section we describe the secondary properties and discuss the impact of their selection on a composition.

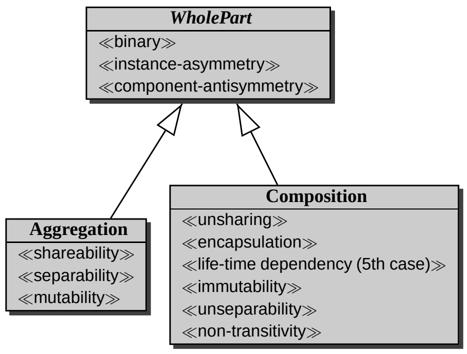


Figure 1. Whole-Part subtypes examples

Encapsulation A component A is encapsulated into a component B when only B can access its services. This implies that: (i) A cannot be part of another component, and (ii) A do not have any relationship of any kind with any component outside of B.

Shareability This property allows a component to be part of several wholes. It can be considered at a *local* level (a part shared among same kinds of wholes – same WPR) or a *global* level (a part shared among different kinds of wholes – different WPRs). The reverse of local shareability is local exclusion. The reverse of total shareability is total exclusion.

Separability and mutability Separability allows a component to be separated from its whole. This property can cause confusion because of its lack of distinction with, in particular, the life-time dependency. This is why we strongly link this property with the one of *mutability*. The mutability is the property which makes it possible to modify the number and/or the identity of the part components of a whole. By opposition, immutability implies that the set of part components of a whole is same throughout its life cycle. It has been formally established that immutability $\Rightarrow$ inseparability and that separability $\Rightarrow$ mutability.

Life-time Dependencies There are nine cases of life-time dependency (between a whole and a part). These nine cases correspond to the combination of ”before/after/same-time” characteristic with the ”birth/death” one. Among them, the *existential* dependency is of particular interest, i.e., the coincidence of birth and of death of the two elements. In this case, the property is directly related to the mutability and separability ones. Indeed, in order to have an existential dependency, it is necessary that the immutability property is selected and thus as well the inseparability one.

Transitivity If a whole component A is composed of a part component B, itself composed of a part component C, the transitivity property consists in making possible that A directly access to C services without going through the B interface.

2.3. Secondary properties: relations and combinations

We have mentioned that certain properties have strong relations between them. It is not our aim to be exhaustive in this paper by describing all these interactions, but we give some examples of them as an illustration of the need of a development environment that support such definitions.

The existential dependency property, for example, is strongly dependent with the one of immutability/inseparability. The property of encapsulation is also illustrative. In our approach, we consider the software composition as a *vertical* one, i.e., we consider any composition as a WPR: a whole component is composed of part components. The latter play the role of service supplier for the whole component. Hence, when the encapsulation property is specified, it implies for the component to be non-shareable locally and globally, since it can provide its services only to its whole component. In the same way, if a subcomponent is not shareable and if its whole component is itself a part component of a third component, then this last will not be able to use directly the services of the first. The non-transitivity property is then implicit. These examples illustrate the importance of the interactions between secondary properties. These interactions make possible to define several subtypes of the WPR (the one that often occur in CBSE). For example (see Figure 1), a relation for which the shareability, separability and mutability properties have been selected will characterize a WPR subtype which is commonly known as *aggregation* (especially for UML users). Another relation possessing the non-shareability, encapsulation, existential dependency, immutability, inseparability and non-transitivity will characterize the subtype called *composition*. We have so far deeply studied only these two subtypes since they are already treated in UML. We currently work at the characterization of other subtypes of WPR, and at the full study of properties interactions. This study is important: (i) to avoid contradictory set of properties, and (ii) to define dependencies between properties.

2.4. Discussions

The main benefit of our approach is the ability to add some precisely-defined properties on the composition relation. It is a significant improvement which has not been incorporated in the last version of the UML notation, which only treat syntactical aspects of composition. It is possible in our approach to specify constraints on the composition itself. It makes possible to consider interactions between components during the modeling phase, early in the life-cycle.

Providing the theoretical framework is not enough. In order to be useful, this framework needs to be supported in some way. A modeling support should allow the expression of the composition properties, and a development support should allow to check and verify the expected properties in a developed system. We are currently experimenting two approaches for this purposes. In an *a priori* approach, based on the definition of a UML 2.0 metamodel, we make it possible to check the soundness of model before final component integration. In a *a posteriori* approach, we provide a development environment for component assemblies that supports our approach. The metamodeling effort is not presented in this paper. In the next sections, we introduce the environment, and we illustrate the concepts presented so far in a concrete case study.

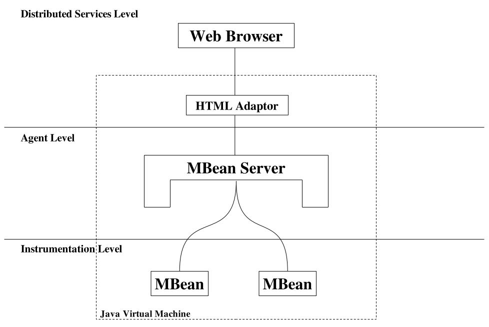
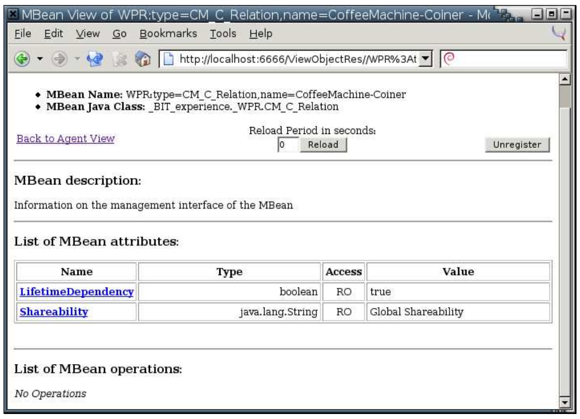


Figure 2. Simplified JMX architecture

3. A verification environment

We now introduce a verification environment based on the WPR. Our environment constrains the assembly of software components, according to the intrinsic properties of the WPR that we have previously defined. We then show how its use on a concrete case study (a coffee machine) can reveal inconsistencies during components assembly.

Our environment uses the JMX (Java Management eXtensions) technology [17]. This library allows the management of Java components, called MBeans (Managed Beans). Each MBean is referenced within a server by a unique name, its *object name*. MBeans can communicate through the MBean server thanks to their object names. Each MBean has a management interface in which it exposes the attributes and the methods which will be made accessible to the other MBeans. Another benefit from this environment is that a Web browser can be connected to the MBean server: the server generates HTML pages to manage the MBeans by accessing their management interface. Figure 2 depicts this architecture and Figure 3 is an example of the manipulation interface. This manipulation interface is completely generated by JMX. We consider this management interface as an example of *configuration interface* as introduced in [4].


MBean View of WPR:type=CM_C_Relation,name=CoffeeMachine-Coiner - M
File Edit View Go Bookmarks Tools Help
http://localhost:6666/ViewObjectRes//WPR%3At

- **MBean Name:** WPR:type=CM_C_Relation,name=CoffeeMachine-Coiner
- **MBean Java Class:** _BIT_experience._WPR.CM_C_Relation

Back to Agent View

Reload Period in seconds: 0 Reload

Unregister

MBean description:

Information on the management interface of the MBean

List of MBean attributes:

Name	Type	Access	Value
LifetimeDependency	boolean	RO	true
Shareability	java.lang.String	RO	Global Shareability

List of MBean operations:

No Operations

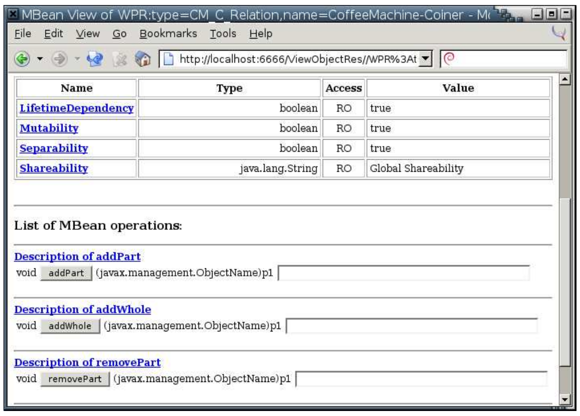
Figure 3. Simple Relation

JMX offers an optional, but very interesting, *relation service*. We have used this service to build an implementation environment. This service allows us to create and to manage relations between components (encapsulated as MBeans). It is used to constrain their consistency (e.g., respect of the specified cardinality). These relations are simple n-ary associations between MBeans in named roles.

The WPR implementation we have realized has been initiated in [8]. It is based on a JMX relation. The properties of WPR are fulfilled by using both the capabilities of the JMX server and its relation service. For instance, the binary nature of the relation between the Whole component and its Part component is ensured by a JMX relation type composed of a *Whole* role and a *Part* role, and the antisymmetry at component level of the relation is checked by an algorithm we have developed which looks through the relation types that are already registered in the relation service.

The architecture of our environment is designed in such a way that one may quickly and easily implement WPR subtypes. We illustrate this process using the two subtypes defined in UML: aggregation and composition. In Figure 1, we have illustrated how aggregation and composition relations can be seen as WPR subtypes. The *WholePart* relation embodies the primary WPR properties and the secondary properties are embodied into *Aggregation* and *Composition*.

In our environment, we have defined a *WholePart* abstract class which implements all the properties of WPR, i.e., primary and secondary ones, and a marker interface for each secondary property. Then to create a new WPR subtype, we extend the *WholePart* abstract class (so the relation inherits the primary properties) and we only use the marker interfaces corresponding to the secondary properties we want to activate. For the *composition* and *aggregation*, this is simply expressed in the following code:


MBean View of WPR:type=CM_C_Relation,name=CoffeeMachine-Coiner - M
File Edit View Go Bookmarks Tools Help
http://localhost:6666/ViewObjectRes//WPR%3At

Name	Type	Access	Value
LifetimeDependency	boolean	RO	true
Mutability	boolean	RO	true
Separability	boolean	RO	true
Shareability	java.lang.String	RO	Global Shareability

List of MBean operations:

Description of addPart
void addPart (javax.management.ObjectName)p1

Description of addWhole
void addWhole (javax.management.ObjectName)p1

Description of removePart
void removePart (javax.management.ObjectName)p1

Figure 4. Mutable & Separable Relation

```
public interface CompositionMBean extends
    Encapsulation,
    LifetimeDependency {
}
```

```
public class Composition
    extends WholePart
    implements CompositionMBean {
    public Composition(...) throws JMException {
        super(...);
    }
}
```

```
public interface AggregationMBean
    extends Shareability,
    Separability,
    Mutability {
}
```

```
public class Aggregation
    extends WholePart
    implements AggregationMBean {
    public Aggregation(...) throws JMException {
        super(...);
    }
}
```

There is no need for additional code. Only the parameters of the constructors are omitted for readability purposes. Besides its use for *a posteriori* verification, this technique offers a great flexibility in building WPR subtypes and gives us a simple way to experiment new relations. So we can practically figure out what are the suitable combinations of properties for software composition, and then define the corresponding WPR subtypes.

Another important aspect of our approach is the complete separation between the component code itself from the composition code. In fact, the composition code is completely supported by our tool. When properties have been set, our environment ensures the respect of the composition properties. Figure 3 was an illustration of the manipulation interface where the specified properties had no additional operation needed. If we consider another example, where the mutability and the separability properties are set for a relation, the tool automatically add in the interface the required operations to change (mutability) and to remove (separability) a component from this relation. In Figure 4, we can see that these properties have been set and that the corresponding relation Web page exposes the necessary buttons to invoke these operations.

4. Implementation in a case study

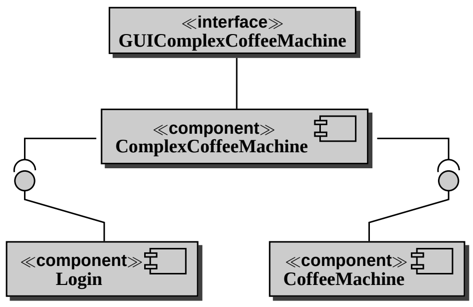


Figure 5. A concrete case study

In this section, we present our case study, then we show the use of our method in this particular case study at modeling level first and at verification level next.

4.1. Introducing our case study

Our case study consists of a drink dispenser system decomposed in three complex components: a *ComplexCoffeeMachine* component, a *Login* component and a *CoffeeMachine* component (see Figure 5). We will only focus

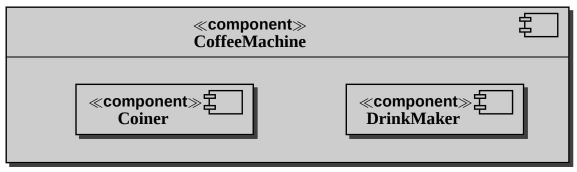


Figure 6. CoffeeMachine details

here on the *CoffeeMachine*. This component is in charge of managing payments, making and delivering drinks. In our approach, this is a *Whole* component. We depict it, using the UML 2.0 formalism [11], as a *PackagingComponent*. It has two subcomponents (or *Part* components) shaped into *BasicComponents*: a component for receiving and giving change, named *Coiner*, and a component for preparing and delivering drinks, named *DrinkMaker* (see Figure 6).

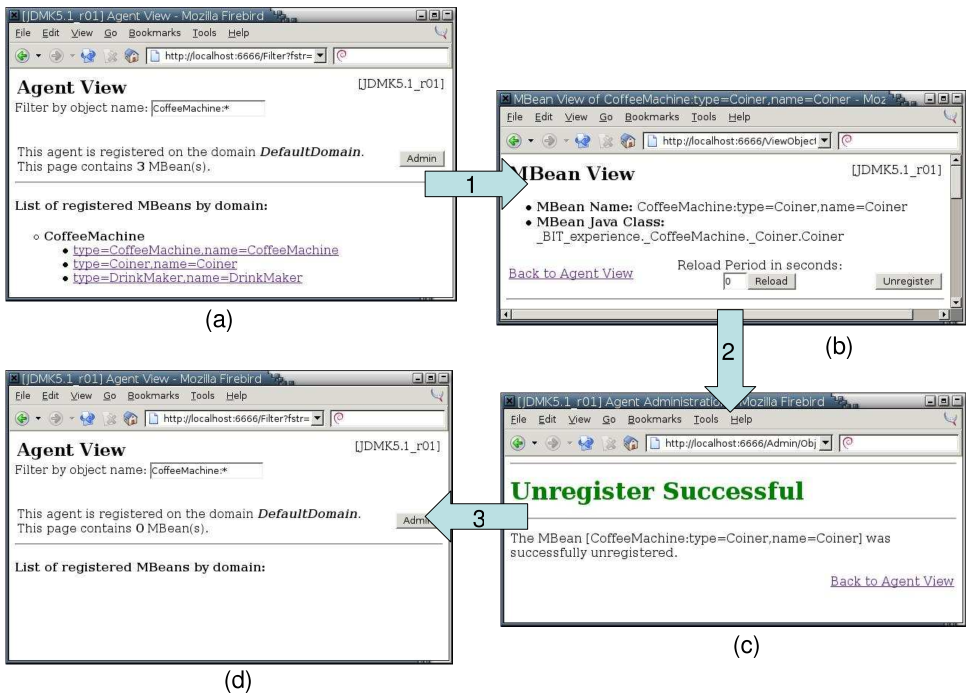
4.2. Use of the WPR properties for the system design

The *DrinkMaker* subcomponent is specific to the *CoffeeMachine* component. It has no other use in the system. Thus, we have specified that only the *CoffeeMachine* can gain access to it. Then *DrinkMaker* is encapsulated in its Whole component, *CoffeeMachine*. To ensure this restricted access, this relationship holds the global and local exclusion property since the *DrinkMaker* can be owned by only one *CoffeeMachine*. To ensure the continuity of the service, the subcomponent must permanently exist. It must not be separable from its Whole and consequently, it also must not be mutable and it is existentially dependent on its Whole.

The case of the *Coiner* subcomponent is different. Indeed, in our case study, the *CoffeeMachine* component is coupled with a graphical interface and a mechanism of electronic wallet. The regular users of the coffee machine can deposit virtual money using the coiner interface of the coffee machine. The electronic wallet is not presented here but it has to use the same *Coiner* component used by the *CoffeeMachine* component. So this component must be shared and it must not be encapsulated into its Whole component: this is the global sharing property. However, the *CoffeeMachine* component must existentially depend on the *Coiner* since we do not want it to be destroyed and let free access to the *CoffeeMachine*.

4.3. Verification

We have developed, for our case study, a Java implementation using our environment. It has been then possible



[JDMK5.1_r01] Agent View - Mozilla Firebird
http://localhost:6666/Filter?fstr=

Agent View [JDMK5.1_r01]
Filter by object name: CoffeeMachine:*

This agent is registered on the domain ***DefaultDomain***.
This page contains **3** MBean(s).

List of registered MBeans by domain:

- CoffeeMachine
  - type=CoffeeMachine,name=CoffeeMachine
  - type=Coiner,name=Coiner
  - type=DrinkMaker,name=DrinkMaker

MBean View of CoffeeMachine:type=Coiner,name=Coiner - Moz
http://localhost:6666/ViewObject

MBean View [JDMK5.1_r01]

- **MBean Name:** CoffeeMachine:type=Coiner,name=Coiner
- **MBean Java Class:** _BIT_experience._CoffeeMachine._Coiner.Coiner

Back to Agent View
Reload Period in seconds: 0

[JDMK5.1_r01] Agent Administration - Mozilla Firebird
http://localhost:6666/Admin/Obj

Unregister Successful

The MBean [CoffeeMachine:type=Coiner,name=Coiner] was successfully unregistered.

Back to Agent View

[JDMK5.1_r01] Agent View - Mozilla Firebird
http://localhost:6666/Filter?fstr=

Agent View [JDMK5.1_r01]
Filter by object name: CoffeeMachine:*

This agent is registered on the domain ***DefaultDomain***.
This page contains **0** MBean(s).

List of registered MBeans by domain:

Figure 7. Lifetime dependency verification

to check the consistency of the properties that we have already defined in the previous section and which are held by the relations between the system components. Depending on the nature of the properties, the environment performs a positive or negative verification: either the environment ensures itself the property to the relation, in this case we are sure that the property holds, or it checks that the property is consistent with the already defined relations and ban this relation if an inconsistency is detected. We will show an example of both possibilities.

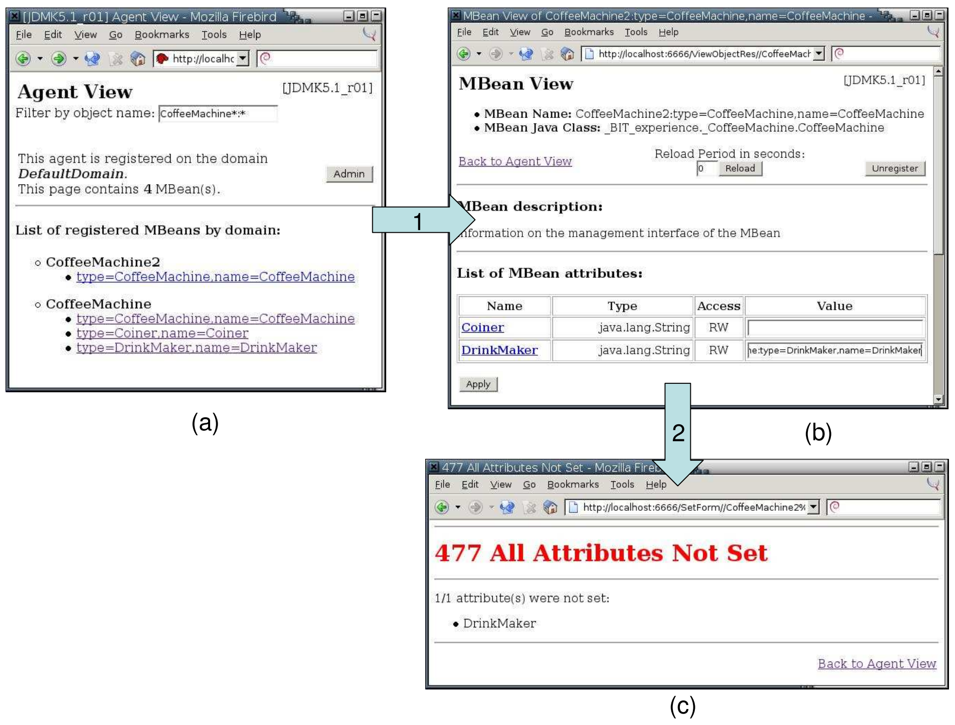
Firstly, in our case study, the *CoffeeMachine* component is in lifetime dependency with the *Coiner* and *DrinkMaker* components. Thus, as an example, the destruction of the *Coiner* component entails the destruction of the *CoffeeMachine* component which in its turn entails the destruction of the *DrinkMaker* component. This is an example of positive verification. Figure 7 depicts this situation. In *(a)*, the three components are manageable through the Web interface. Once the *Coiner* component is selected, it is removed in *(b)*, which is validated by JMX in *(c)*. Then, none of the three components are manageable any longer through the Web interface in *(d)*.

Secondly, we have specified that the *DrinkMaker* component is in exclusive relation with the *CoffeeMachine* component. Thus, this relation is not shareable (i.e. global and local exclusion). For instance, another component which has the same type as the *CoffeeMachine* should not be in relationship with the *DrinkMaker* component. Figure 8 depicts this situation where the violation of the property notified in *(c)* prevents the *DrinkMaker* to be bound to another *CoffeeMachine* component.

5. Conclusion and future works

We have presented in this paper a modeling approach aiming to improve CBSE, and more precisely composition. The basic idea is to express a composition as a particular case of whole-part relationship. We have shown, using a set of well-defined properties, how we can specify the semantics of software composition. We have then detailed a tool, based on an existing Java library, that allows us to define, and later to check, the properties of a composition. We illustrated our approach using a particular case study.

The work presented in this paper is integrated in an overall project aiming at the definition of a complete environment for development of component-based systems, from the modelization, to the test of assemblies of components. The modeling support presented in this paper is not currently supported by any UML tool. We are currently defining a UML profile implementing our properties definitions. This profile will be integrated into a UML tools. We are currently exploring two possible supports, a public domain


[JDMK5.1_r01] Agent View - Mozilla Firebird

Agent View [JDMK5.1_r01]

Filter by object name: CoffeeMachine*:*

This agent is registered on the domain *DefaultDomain*.
This page contains 4 MBean(s).

List of registered MBeans by domain:

- CoffeeMachine2
  - type=CoffeeMachine,name=CoffeeMachine
- CoffeeMachine
  - type=CoffeeMachine,name=CoffeeMachine
  - type=Coiner,name=Coiner
  - type=DrinkMaker,name=DrinkMaker

MBean View of CoffeeMachine2:type=CoffeeMachine,name=CoffeeMachine -

http://localhost:6666/ViewObjectRes//CoffeeMach

MBean View [JDMK5.1_r01]

- MBean Name: CoffeeMachine2:type=CoffeeMachine,name=CoffeeMachine
- MBean Java Class: _BIT_experience._CoffeeMachine.CoffeeMachine

Back to Agent View

Reload Period in seconds: 0

MBean description:

nformation on the management interface of the MBean

List of MBean attributes:

Name	Type	Access	Value
Coiner	java.lang.String	RW	
DrinkMaker	java.lang.String	RW	ie:type=DrinkMaker,name=DrinkMaker

477 All Attributes Not Set - Mozilla Firebi

http://localhost:6666/SetForm//CoffeeMachine2%

477 All Attributes Not Set

1/1 attribute(s) were not set:

- DrinkMaker

Back to Agent View

Figure 8. Shareability verification

one, SMW [13], and a commercial one, Objecteering [14].

As we have illustrated, the lack of effective and CBSE-dedicated support (in terms of modeling) is one of the main obstacle to the massive use of software components, and in particular of Commercial Off-The-Shelf (COTS) components. We have started a three years project aiming at the definition of such a dedicated support. The main goal of this project is the definition of a Component Modeling Language (CML). Based on the general modeling notation UML, and incorporated specific concepts such as those provided by Architecture Description Languages (ADLs), this project enables us to federate our efforts around composability (like those described in this paper) and those around behavior prediction (and more precisely extra-functional properties prediction) of assemblies of components [7].

References

[1] ARTIST. Component-based Design and Integration Platforms : Roadmap. Technical Report W1.A2.N1.Y1, Project IST-2001-34820, 2003. Available at http://www.artist-embedded.org.

[2] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel. Aggregation and Composition in UML: A Rectifi ed Implementation Based on Whole-Part Theory. *IEEE Transactions on Sotfware Engineering*, 29(5), 2003.

[3] N. Belloir, J.-M. Bruel, and F. Barbier. Whole-Part Relationships for Software Component Combination. In G. Chroust and C. Hofer, editors, *Proceedings of the 29th Euromicro Conference on Component-Based Software Engineering*, pages 86–91. IEEE Computer Society Press, Sept. 2003.

[4] J. Bosch. *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*. Addison-Wesley, May 2000.

[5] J. Bosch, C. Szyperski, and W. Weck. WS5. The Eighth International Workshop on Component-Oriented Programming (WCOP 2003). Workshop report, 2003.

[6] J.-M. Bruel and I. Ober. The new UML 2.0 Component Model: Critical View. In E. Grosspietsch and K. Klöckner, editors, *Proceedings of the Work in Progress Session at the 29th Euromicro Conference*, Sept. 2003.

[7] O. Constant, W. Monin, B. Rougeot, and F. Barbier. Performance et composants logiciels : une démarche en conception. In *Submitted at JC'2004*, Mar. 2004.

[8] Fabien Roméo. Composabilité des composants logiciels et test intégré. Technical report, Mémoire de DEA Programmation et Système, Université Paul Sabatier, Toulouse III, Toulouse, 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] OMG. MDA Guide V1.0.1. OMG document – omg/03-06-01, Object Management Group, jun 2001. Available at http://www.omg.org/mda/.

[11] OMG. UML 2.0 Superstructure Final Adopted specifi cation ptc-o3-08-02. OMG document, Object Management Group, Aug. 2003.

[12] D. Petit, G. Mariano, and V. Poirriez. Génération de composant à partir de spécifi cations B. In *Proceedings of the conference "Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)"*, Rennes, France, january 15-17, 2003.

[13] SMW UML tool. http://www.abo.fi /˜iporres/html/smw.html.

[14] Softeam. Objecteering uml tool. Available at http://www.objecteering.com/.

[15] H. C. Software and S. C. Group. High confi dence software and systems research needs. Technical report, Interagency Working Group on Information Technology Research and Development, january 2001. Available at http://www.ccic.gov/pubs/hcss-research.pdf.

[16] J. A. Stafford and K. Wallnau. *Building reliable component-based software systems (Ivica Crnkovic and Magnus Larsson editors)*, chapter Component Composition and Integration, pages 179–191. Artech House Publishers, Boston, 2002.

[17] Sun. Javatm management extensions (jmx) 1.2.1, 2003. http://java.sun.com/products/JavaManagement/.

[18] P. S. with Rob Pooley. *Using UML: software engineering with objects and components*. Object Technology Series. Addison-Wesley Longman, 1999. Updated edition for UML1.3: fi rst published 1998 (as Pooley and Stevens).