Component Based Development : a Composition Oriented Approach

Nicolas Belloir
University of Pau – LIUPPA
BP 1155, 64013 Pau CEDEX – France
Email: Nicolas.Belloir@univ-pau.fr

Jean-Michel Bruel
University of Pau – LIUPPA
BP 1155, 64013 Pau CEDEX – France
Email: Jean-Michel.Bruel@univ-pau.fr

***Abstract*— The growing complexity of Information Systems, as well as the adaptability and the flexibility constraints they require, call for for the use of components for their development. A dedicated software engineering approach - Component-Based Software Engineering (CBSE) - has emerged. Our view is that composition is the critical point of such development. By composition, we mean semantical integration of components. We present in this article a composition oriented approach for component-based development. Composition is treated by a specific relation based on Whole-Part relationship. This relationship can be characterized by formally specified properties and allows for constraint structural and dynamic relations between components. This relation is realized by a composition framework which we have developed, consisting by a UML profile implementing the composition relationship and a deployment environment ensuring the respect of the composition properties.**

I. Introduction

In Software composition components are assembled in order to form a new system or a higher granularity component. Component models, such as CCM or .NET, provide a recognized support for the development of such systems. However, software engineering technologies and methods, specific to this domain, are not enough. Component-Based Software Engineering (CBSE) requires more research efforts in theses directions [1]. In this context, composition is considered to be a fundamental basis. Indeed, composition can be considered at different level (analysis, design, assembly ...) during application lifecycle. Previous research works mainly focused on composition at assembly level. It is now recognized that considering composition in earlier levels in the application lifecycle is a priority in order to design trusted applications. Numerous current works as [2] are investigated this way. In this context, the formalization of the composition is really important. Design technologies lack dedicated and well formalized constructions. In particular, UML, which is the *de facto* modelling standard, does not provide features allowing to design CBSE concepts. The latest version of this language (2.0) is more suitable for CBSE but retains many incoherences in semantics of the CBSE concepts.

Based on our on-going research on component composition [3], this paper describes a development approach using a composition model to formalize composition relationships and a composition framework allowing design, development and management of composition relationships. The remainder of paper is structured as follows: section II describes some of the problems related with composition. Section III presents a composition model based on Whole-Part relationship, proposing a modification of the UML metamodel allowing definition of composition properties at conceptual level. We explain in particular how our model can be used in existing component based development methods. Finally, section IV proposes a composition framework supporting our composition model.

II. Component composition in development process

Component integration (called component assembly) is not the same as component composition. According to [4], integration is the "mechanical task of wiring components together" while composition is "the ability to assign properties to the whole based on the properties of the parts and the relationships between the parts" so that it "supports the possibility of determining the properties of assemblies in order to check their run-time compatibility". Thus, composition focuses on the semantical aspect of assembly. An application built by component integration is not necessarily viable at running time. Determining valid component combinations implies consideration of both architectural and semantical dimensions during component integration.

We have defined two kinds of composition relationships: *horizontal* composition deals with service exchange between components of same granularity. This relationship is the most used one to integrate components. *Vertical* composition concerns relations between a lower (*subcomponent*) and a higher (*composed*) granularity component. One could refer a component composed from subcomponents, or about hierarchical composition (as Fractal [5] for example). A component uses the services of its subcomponents in order to provide itself more complex services. During deployment, a component is configured in order to be adapted to its new environment. Its subcomponents can be recursively configured but this configuration must be managed by the component itself. Component provided services can be implemented by the component itself, or by one of the subcomponents, or by a combinaison of both component and subcomponents functionalities. Incrementing coupling between components is one of the negative consequences of vertical composition, depends of the manner in which the composition link is implemented. However, this

approach allows us to organize application architecture in well-identified groups and so reduce application complexity. Replacing one subcomponent may only have a direct impact on the component containing it. At conceptual level, considering composition allows better specification of component relationships, using specific properties such as dependency between components. It is particularly important, for instance, for components adaptation [6]. Composition design is even more important with black-box components, since they cannot be adapted by definition. So, it is necessary to determine the composition properties to improve the compatibility between components at runtime [4].

At conceptual level, some studies try to better define semantics of vertical composition. For instance, the *Composite* pattern [7] organizes entities to be designed under hierarchical structure in which entities and compositions of entities are treated at the same level. Another approach uses design contracts [8] to specify interaction between components. The ACCORD project [9] has defined principles which allow for the building of component software architectures based on both components and contracts concepts. The formal method community has also studied composition [10]. Formal approaches offer a precise semantics but are less accessible. On the other hand, graphical notations such as UML are often criticized by the scientific community because of their "lack" of semantics. However, they offer a readily approachable formalism, well-known by the industry, which has proved its usefulness in software development [11]. Using both approaches provides a better specification. Some approaches propose using a formal method with a graphical notation to solve this problem. Our proposition is inverse. We want to determine if we can improve the semantics of semi-formal approaches by constraining their syntax using languages such OCL [12]. UML is the most used semi-formal approach in CBSE and uses the OCL formal language to describe it semantics. We apply to UML our work on the improvement of the semantics of the composition relationship at conceptual level.

When speaking about component-based development methods, it is necessary to differentiate between design *for* reuse and design *by* reuse. Our work, even if dedicated to a transversal problem, is more concerned with design by reuse, *i.e.* building application by composition of reusable components composition. In such process, we encounter the same lack of consideration as in component composition. Methods generally focus more on components themselves or on relationships between components rather than the precise semantics for these relationshipsThey often use UML as design support. In [13], composition is only mentioned at assembly level and there is no precise clarification on this topic. In the OPEN [14] proposition, hierarchical composition is treated with the concept of *Containment*. It is limited to a few aspects of the possibilities offered by composition relationships. Lack of composition semantics can also be encountered in design graphical support as well as in development methods. For this reason our work fundamentally addresses simultaneous design support and methods. However, we do not neglect its implementation across the development processes used in component-based development, as we show it in the following sections.

III. A COMPOSITION MODEL

A. A composition model based on Whole-Part Relationship

In previous work [15], Whole-Part Relationship was used as a semantical basis to characterize UML aggregation and composition, in oriented-object design. We have systematically reviewed the properties identified in [15]. We have studied both the value of each property in the component world and the relationships among this properties. Both *asymmetry at instance level* and *antisymmetry at type level* properties are primary properties: all composition relationships must respect these properties. The remainder consists of a set of characterization properties which allows the derivation of specialized types of composition relationships. Moreover, we think that UML *composition* and *aggregation* relationships are not enough to design all possible combinations at the architectural level. Indeed, UML standard *composition* relationship deals with *encapsulation* and *unshareability* properties and *aggregation* with *non encapsulation* and *shareability* [15]. We propose to define four composition relationships, allowing more precise design at architectural design. Tab I shows these four relationships and their characterization properties.

	Strong Composition		Lightweight Composition		Strong Aggregation		Lightweight Aggregation	
	stat.	*dyn.*	*stat.*	*dyn.*	*stat.*	*dyn.*	*stat.*	*dyn.*
encapsulation	yes	yes	yes	yes	no	no	no	no
shareability	no	no	yes	yes	no	no	yes	yes
separability / mutability	no	yes	no	yes	no	yes	no	yes
lifecycle dependency (case number)	1	4	1	4	9	9	9	9

TABLE I

DIFFERENT KINDS OF COMPOSITION RELATIONSHIP

B. Application to UML

Our model is based on UML 1.x, waiting for a stable release of UML 2.0. Indeed, from a notational point of view, UML 2 provides a pertinent response to the weakness of UML 1.x [16]. However, we think that these improvements are not enough. Our approach could reuse the notational improvements but if we consider the specification of the links between a Composite and its Parts, our profile remains always a valid and indispensable approach. The concept of composite can be translated as a *PackagingComponent* and a Part can be viewed as a *BasicComponent*.

The weakness of UML 1.5 are listed here : support to component assembly, concept of plug-substitutability, specification of component interaction patterns, design of the component

runtime context, definition of profiles for specific architectural models. The new UML standard responds partially to these exigences. However, we believe that it doesn't respond to the first point which was linked to composition [17] : (i)improvement of the notation semantics (specially of the associations) to support component based development ; (ii) better support for interfaces and (iii) new protocols for interface formalization.

Because of the lack of experience feedback, we have developed our private heuristics about using these new concepts (see also [18]) and we preferred to focusing on *aggregation* and *composition* relationships rather than on new concepts as *composite structures* because of their lack of precise semantics [19]. Our proposition to modify the UML metamodel is based on a new abstract metatype representing the composition relationship between two components and describes a design concept linked to a particular type of *classifier*. It makes more complex the UML language and lowers it genericity. However, this branch of our model will eventually be able to be inserted in a stable version of UML 2.0 as a generalization of a generic relationship. We think that a specific relationship for component composition is very important in order to support a more intense usage of this paradigm.

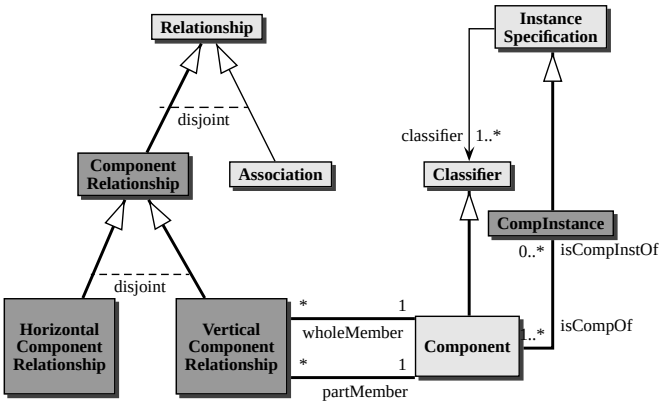

Fig. 1. New metaclasses for the composition relationship

Figure 1 shows our metamodel. The new metaclasses and the added links are in dark grey. The new abstract metaclass *Component Relationship* represents all kinds of relationship between components. The vertical and horizontal relationships are respectively represented by the *Vertical Component Relationship* and the *Horizontal Component Relationship* metaclasses. These two metaclasses are disjointed and implement the *Component Relationship* metaclass.

In contrast to the first version of our metamodel [3], the characterization of the *Whole* or the *Part* component notion is translated by the component *role* in the relationship between association links and the *Vertical Component Relationship* metaclass. The *InstanceSpecification* metaclass is specialized in the *CompInstance* metaclass in order to keep an existing UML 1.x notion that has disappeared in UML 2.0. The composition properties identified in the Tab I characterize the *Vertical Component Relationship*. Optional properties are included as its attributes.

The four kinds of composition relationship are included in the metamodel with four metaclasses. Each one inherits from *Vertical Component Relationship*. Figure 2 shows this part of the metamodel. We have translated the properties as stereotypes. They express specific value of the metaattributes and allow to specialize metaclasses from stereotypes values. All relationships must implement the primary properties that are defined in the *VerticalComponentRelationship* metaclass.

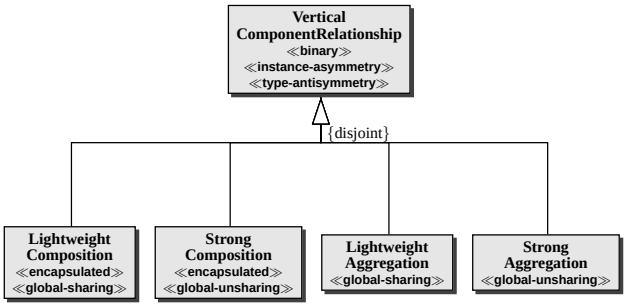

Fig. 2. Representation of the different kinds of composition relationship

The composition properties have been translated in OCL rules in [15]. The semantics of the composition properties in the component world is globally the same. The OCL rules of our metamodel are only contextual adaptations of the previous OCL rules. In order to illustrate an OCL rule, the following one translate the *irreflexivity* property which is necessary to specify *asymmetry at instance level*:

```
1 context VerticalComponentRelationship inv
   Irreflexivite :
2    wholeMember.compInstance->forAll(w | w.oclIsKindOf
       (partMember) implies not w.part->includes(w))
3    partMember.compInstance->forAll(p | p.oclIsKindOf(
       wholeMember) implies not p.whole->includes(p))
```

C. Using composition model

The most frequently quoted disadvantage is the lack of support for the analysis and the validation of component compositions, both at structural and behavioral level. We believe that the reason is the lack of precise semantics in composition relationships. Our approach does not seek to generate a new method, but rather to improve, by both formalization and tools construction, support for essential problem of composition. It is however necessary to further explain the use of our model, and we chose to speak about both the cycle of component oriented development used with UML [13].

Component oriented development processes generally provide little assistance in the verification of the conformity of implementation to its specifications. For instance, in the Cheesman & Daniels approach (*UML Components*), assembly description occupies just one page. And similarly for deployment in runtime environment [13, p. 162-164]. The contribution of our model to this process focuses precisely on providing support in these critical activities. By reinforcing both specification of components and their architecture, through the structural and behavioral composition relationships, we help the developer in this particular stage of the

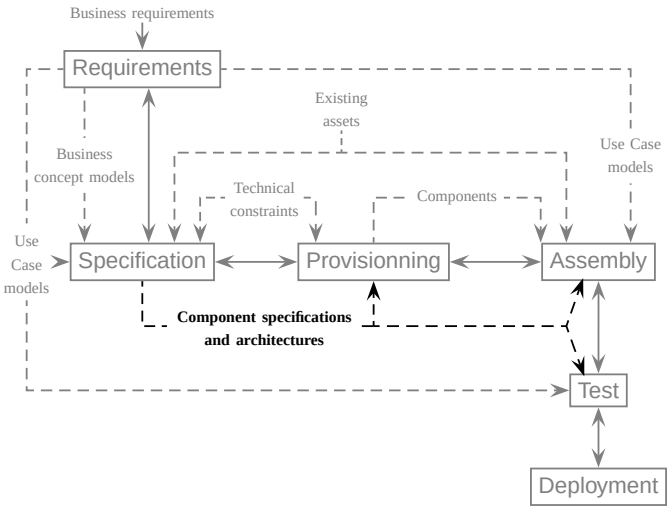

Fig. 3. "UML Components" development cycle [13, p. 27]

lifecycle (cf. Figure 3 – bold part). Our model clearly deals with the management of composition relationships between components, from advanced design to implementation.

IV. A COMPOSITION FRAMEWORK

In this section, we present our composition oriented framework which provides concrete arguments supporting what we have just enunciated, showing the practicability and usefulness of our model. It was elaborated from two ideas: (i) focusing on composition during all development cycle is important, and (ii) one of the problems related to the introduction of new notions at model level is that these notions seldom have a direct translation into programming abstractions. The engineer in charge of the realization of models must therefore arbitrarily choose a means of implementation for each model concept with existent programming abstractions. Our framework attempts to address these constraints. It is composed from both a UML profile, integrating our composition model, and a deployment environment, integrating the paradigm of the composition relationship previously defined.

A. A UML composition profile

A UML profile [20] allows the adaptation of the semantics of UML without changing the metamodel, with well-defined extension mechanisms, consisting of the concepts of *stereotype*, of *tagged value* and of *constraint*. We use these mechanisms to implement our composition model. We have developed our profile with the *Profile Builder* module of *Objecteering* [21], an UML modeler. In our opinion, it is the only software providing support for UML profiles development.

1) Description of the profile: Our profile, called *WPCP* (Whole Part Composition Profile), allows to specify application in terms of both type and instance of components. Thus, we use two kinds of specific class diagrams. The Components Type Diagram describes the architecture design with types of components. The Components Instance Diagram instantiates the first diagram and allows architecture design with instances of components. Components and components instances are translated by the stereotypes inheriting from the *Class* metaclass: <<*WPCPComponent*>> and <<*WPCPCompInstance*>>. Composition relationships are translated by the <<*VCR*>> (Vertical Component Relationship), <<*SCOMP*>> (Strong Composition), <<*LCOMP*>> (Lightweight Composition), <<*SAGG*>> (Strong Aggregation) and <<*LAGG*>> (Lightweight Aggregation) stereotypes that are specializations of the *Relationship* relationship. The <<*Whole*>> and <<*Part*>> stereotypes characterize the two ends of the relationship. Tagged values represent metattributes of the metamodel. In our model, only the *VerticalComponentRelationship* metaclass, and those inheriting from it, have attributes. Finally, composition properties are expressed by tagged values.

The profile provides some scripts, based on the composition properties, used to verify the semantics of models. These scripts were realized in the inner *Objecteering* language, *J language*, because the software does not allow to implement OCL rules. We translated the OCL constraints about composition properties in the *J language*. For example, the following code verifies that the relationship is a binary relationship:

```
1 boolean Association::checkRolesOfRelation () {
2 // Check relationship size
3 if (ConnectionAssociationEnd.size != 2) {
4     StdErr.write("ERROR on ", Name, " relationship - A
        Composition relationship must be binary : ", NL)
      ;
5     return false;}
6 return true; }
```

2) Illustration of WPCP: We illustrate here the use of our profile with the coffee machine case study. The coffee machine is composed of a drink maker and a coiner. It also provides an electronic purse. Money is added by the coiner. This example shows the combination of three composition relationships: COFFEEMACHINE - *Coiner*, COFFEEMACHINE - *DrinkMaker* and E-MONEY - *Coiner* (we use capital letters for Whole). In the component type diagram, we created four components: *CoffeeMachine*, *Coiner*, *DrinkMaker* and *E-Money*. We then created the relationships between these components by drawing associations between components. We set the stereotype corresponding to the desired kind of relationship with the properties window. Figure 4 shows a screen capture of the realized model. When the type model is specified, the designer would run a script to verify validity of the realized model against the composition model.

When the component type diagram is validated, the designer specifies the application with component instances realizing the component instance diagram, and verifies its conformity with the component type diagram. Firstly, the designer must create components instances. The number of instances depends of the specified relationships in the corresponding component type diagram. For example, the relationship between *CoffeeMachine* and *Coiner* is a *SCOMP*. It implies the encapsulation and global exclusiveness properties. However, the

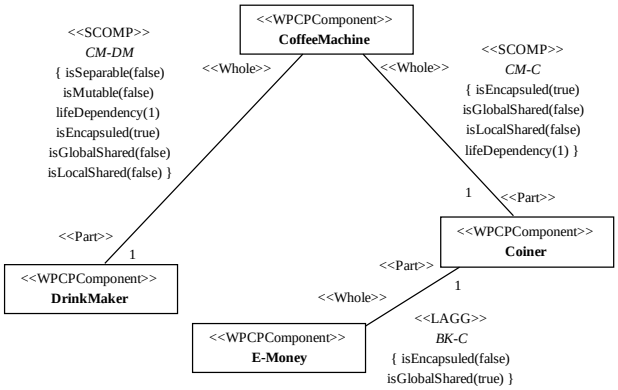

Fig. 4. The component type diagram generated by the WPCP profile

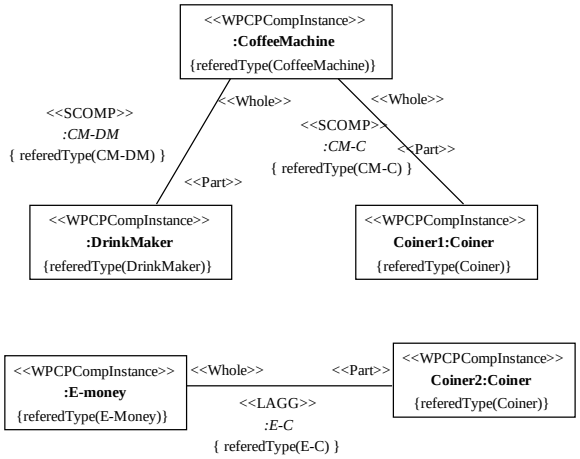

Fig. 5. The component instance diagram generated by the WPCP profile

E-Money component also has a relationship with the *Coiner* component. Thus, the global exclusiveness property implies two instances of the *Coiner* component: one (*Coiner1:Coiner*) has a relationship with the instance of *CoffeeMachine*, the other (*Coiner2:Coiner*) has a relationship with the instance of *E-Money*. After the instances specification, the designer must draw the relationships between the instances. Figure 5 shows a screen capture of the generated diagram. When the component instance diagram is finished, it is necessary to verify with *J language* scripts its compliance with the corresponding component type diagram (Objecteering tool has no dynamic verification) and its specified composition properties.

B. A deployment environment for composition

In practice, conformity between the developed software and its corresponding specification is unfortunately not systematically guaranteed. One of the reasons is that deployment environments do not provide means to directly implement all the concepts expressed at model level. From this observation, we developed a composition environment called Whole-Part Composition Environment (WPCE), allowing both Java components and their relationships management, based on postulates enunciated in the previous section, and on our composition model on the other hand. A specific environment for composition allows an *a priori* verification of the compliance of the code facing its design models. Indeed, if a composition model declares that composition has one property, and if the composition environment allows the developer directly implement at assembly level this property, then the respect of this property is ensured by definition through the environment. In a pure formal approach, it would be necessary to prove the validity of such property.

1) A Java composition environment for a rigorous development: WPCE is using the JMX [22] technology allowing management of Java components, called MBeans (Managed Beans). JMX permits connection of a Web browser to an Mbeans server which provides some Web pages from which it is possible to manage MBeans by accessing their management interfaces. The JMX relation service defines classes allowing the construction of relationships between MBeans components, and centralizes all the operations on these relationships to support their consistency. The operations can be made accessible through a Web browser with any other MBean.

Our proposition firstly involved transforming a component in a MBean component. Then, we changed the JMX relation service so that it offers the specification of composition properties. Designed architecture allows us to quickly and easily implement the sub-types of composition relationship. By simplification, we employ the brief names of these sub-types. The offered implementation follows a similar model to specialize these relationships. Properties are translated by interfaces. Classes that implement the sub-types of composition relationships extend all the relationship properties. The following code illustrates this for the *SCOMP* class:

```
1 public class SCOMP extends WholePart implements
   SCOMPMBean {
2     public SCOMP(...) throws JMException {
3         super(...);
4 }}
```

This code is very simple. Here, only constructor parameters are hidden. The code is similar for all sub-types. The differences between sub-types are only located in the MBean interface that they implement. This interface is constituted from the interfaces of the properties characterizing the type of the relationship.

2) Illustration: We illustrate the use of our environment with reference to the coffee machine case study. In order to present a relevant example, we have chosen here to set the COFFEEMACHINE - *Coiner* relationship globally unshareable and not encapsulated by the *CoffeeMachine*. According to the nature of the property, the environment performs a negative or positive checking : either the environment vindicates the property of the relationship itself (by implementation), and in that case the property is assumed to be proven, or the

environment proves that the property of the relationship is consistent with already defined relationships, and in this case it prohibits this relation if a contradiction is found. We illustrate a violation of the unshareability between the *DrinkMaker* and the *CoffeeMachine* components. Let consider there are only two instances of *CoffeeMachine*. One has a relationship with *Coiner* and *DrinkMaker*. The second has no relationship with any component. One tries to instantiate a new relationship between the *DrinkMaker* component, already used by the first *CoffeeMachine* component. The environment detects the violation of the unshareability property and starts an exception.

C. Toward automatic model transformation from profile to composition environment

Even if the previous composition environment allows to directly express composition properties, most of the final architecture is based on JMX and is globally repetitive. Actual research trends seek to automatically generate specific model for the final architecture. In this context, we are working on our framework to automatically transform realized composition diagrams, that are independent of any technical support, in specific platform composition diagram. Thus, this specific composition diagram can be finally transformed into code skeleton. This approach is influenced by the Model-Driven Engineering (MDE) approach promoted by the OMG [23]. We firstly specified and realized a profile allowing the expression of a WPCE architecture. Secondly, we identified transformation rules for translating composition models into WPCE composition models. Thus, we implemented scripts preforming this transformation, and hence validates all composition properties. Thirdly, we implemented scripts generating WPCE code skeleton. Because of the limited place of the paper, we do not describe more precisely this approach here.

V. Conclusion

In CBSE, the composition concept is central since it characterizes the rules of assembly between components. However, it is often considered too late in the application lifecycle. In this context, we have presented an approach that focusses on composition along all the development lifecycle. Our approach uses both a UML metamodel introducing a new vertical composition relationship devoted to components composition and some composition properties formalized in OCL. We have shown the feasibility of this approach by implementing a composition framework based on two tools: a UML profile for the conceptual level and a composition deployment environment for the software level. Both tools implement the composition properties of the composition model as executable constraints. They guarantee respect of these constraints by both the design models and the final application.

Our work is consistent with the actual trends that give more and more significance to models in the development of applications. It implies the use of (semi-)graphical languages in which the elements of notation have a precise meaning. In this context, we worked to increase significantly the power of expression of composition in UML and contributed to specify its semantics, that is to say to constrain its syntax [11].

We are exploring two perspectives of these studies. The first one consists of inserting into our approach the use of the Fractal component model [5] which allows hierarchical compositions. It is built on implicit properties among which some are comparatively close to ours. On the other hand, we are trying to automate our composition approach by using models transformation principles. The idea is to generate from the conceptual diagrams independent of any technological support, the code skeleton of the application in the specific composition environment.

References

[1] High Confidence Software and Systems Coordinating Group, "High confidence software and systems research needs," Tech. Rep., january 2001. [Online]. Available: http://www.ccic.gov/pubs/index.html

[2] J. Dong and S. Yang, "Towards Trusted Composition in Software Design," in *Proc. of the Eighth IEEE Int. Symposium on High Assurance Systems Engineering (HASE'04)*, Tampa, USA, 2004.

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

[4] J. A. Stafford and K. Wallnau, "Component composition and integration," in *Building reliable component-based software systems*, I. Crnkovic and M. Larsson, Eds. Artech House Publishers, 2002, pp. 179–191.

[5] E. Bruneton, T. Coupaye, and J. B. Stefani, "Recursive and dynamic software composition with sharing," in *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, June 2002.

[6] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing Adaptive Software," *IEEE Transactions on Software Engineering*, vol. 37, no. 7, pp. 56–64, July 2004.

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

[8] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1997.

[9] "Accord Project," http://www.infres.enst.fr/projets/accord/.

[10] J. R. Kinitry, "Semantic Component Composition," in *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, Oct., 6-8 2002.

[11] D. Harel and B. Rumpe, "Meaningful Modeling : What's the Semantics of "Semantics" ?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[12] OMG, "Object Constraint Language Specification v.1.1," Object Management Group," OMG document, Sept. 1997.

[13] J. Cheesman and J. Daniels, *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[14] D. Firesmith, B. Henderson-Sellers, and I. Graham, *Open Modeling Language - Reference Manual*. Cambridge University Press, 1998.

[15] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel, "Formalization of the Whole-Part Relationship in the Unified Modeling Language," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 459–470, May 2003.

[16] C. Bock, "UML 2 Composition Model," *Journal of Object Technology*, vol. 3, no. 10, Oct. 2004.

[17] T. Weigert, "Uml 2.0 rfi response overview," http://www.omg.org/docs/ad/00-01-07.ppt, 1999.

[18] S. W. Ambler, "The official agile modeling site – the diagrams of uml 2," Available at http://www.agilemodeling.com, 2003.

[19] J.-M. Bruel and I. Ober, "Uml 2.0 composition support," *Studia Journal*, vol. LI, no. 1, pp. 79–99, june 2006.

[20] OMG, "White Paper on the Profile mechanism - version 1.0," Object Management Group," OMG document, Apr. 1999.

[21] Softeam, "Objecteering software," Available at http://www.objecteering.com/.

[22] S. Microsystems, "Java management extension," Available at http://java.sun.com/products/JavaManagement/.

[23] J. Bézivin and O. Gerbé, "Towards a Precise Definition of the OMG/MDA Framework," in *Proceedings of the Conference on Automatonous Software Engineering (ASE'01)*, Nov. 2001.