From Composition to Connectors

Manzoor AHMAD*, Jean Michel-BRUEL*, Antoine BEUGNARD**
*Institut de Recherche en Informatique de Toulouse (IRIT) Université Paul Sabatier, 118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9, France
*IUT-Blagnac 1, Place Georges Brassens BP 60073 31703 Blagnac Cedex
** INSTITUT Telecom/TELECOM Bretagne, CS 83818, 29238 Brest Cedex 3, France
Contact: bruel@irit.fr

ABSTRACT

Composition is central to Component Based Software Engineering [1]. On one hand, we can express the composition of Whole Part Relationship (WPR) with the help of some properties [2], recognized as WPR properties based on which a UML profile is being devised called WPCP (Whole Part Composition Profile). On the other hand, software composition can be achieved with the help of a connector acting as glue between the communicating components [3]. However, these two approaches consider composition at different levels as WPR properties deal with it as a vertical one with the Whole and the Part components at different granularity levels while Software connector considers it as a horizontal one with the connector in the middle and the communicating components on both sides. In WPR, a precise description of the properties is given which is being undermined by the lack of implementation support. The software connector approach is a promising approach for the implementation of the connectors but it lacks the description of connectors. This paper is a contribution towards bridging the gap between the two approaches. In order for the two approaches to work together, and taking the benefits of the WPCP in code generation of software connectors, as a first step, we match the WPR properties with software connector.

KEYWORDS

Whole Part Relationship (WPR) properties, Connectors, Whole Part Composition Profile (WPCP)

1. INTRODUCTION

CBD (Component-Based Development) is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manner. Constructing an application under this new setting involves the composition/assembly of prefabricated, reusable, independent pieces of software called components [5]. Szyperski [4] indirectly states that components have to be composed to work together in order to build a system. The ultimate goal is to be able to reduce developing costs and efforts, while improving the flexibility, reliability, and reusability of the final application due to the reuse of software components already tested and validated. Component oriented programming aims at producing software components for a component market and for later composition. This requires standards to allow independently created components to interoperate, and specifications that put the composer into the position to decide what can be composed under which conditions [6].

In this context, building an application is based on a process to compose/assemble plug & play components. Therefore, it requires the following phases: on one hand, the description and implementation of plug & play components is needed (*specification & implementation CBD phases)*; on the other hand, a process to interconnect and deploy components is required (*package, assembly and deployment CBD phases)*. CBD methodology increases the quality of software by providing flexibility, adaptability and reusability through the assembly/composition of independent software components [5].

In WPR approach any composition is seen as a high level component (Whole) composed of subcomponents (Parts). The Whole component provides more elaborated services and is based on the services of its Part components. This work is dealing with one of the limitations of the current approaches supporting CBSE, i.e. software composition is only treated at assembly time. This is the main cause of concern for adaptability, which is difficult to achieve when the dependencies between components are not formally specified. So the main focus is concentrated towards component composition as early as in the development process.

A connector is a reification of an interaction, a communication or a coordination system. At an abstract level, it exists to serve the communication and coordination needs of unspecified interacting components. Later in the life cycle, it describes all the interactions between special components by relying on the own interface specifications of these components. It also offers application independent interaction mechanisms like security and reliability.

In the remainder of this article, section 2 describes the background and the two approaches. In section 3, we analyze the link between the two approaches, the technical conclusion and the

lessons learned. Section 4 concludes the paper with future extensions.

2. BACKGROUND

On one hand, we can express the composition of the WPR (Whole Part Relationship) with the help of some properties [2], recognized as *WPR properties* based on which a UML profile is being devised called WPCP (Whole Part Composition Profile) [7]. On the other hand, software composition can be achieved with the help of a *connector* acting as glue between the communicating components [3].

2.1. COMPOSITION PROPERTIES

The theoretical framework is based on an adaptation of the formalization of the WPR to CBSE (Component Based Software Engineering). This work has extended the semantic properties of the WPR by adapting its formal base to software composition. The authors in [2] have determined which properties to apply to component composition and defined formally these properties.

Dependency variations between a whole and its parts may be expressed with the help of a precise set of basic characteristics. Some of these are always present in a WPR and some are optional, on the basis of which they are split into two categories: primary properties (properties that a WPR must always respect) and secondary properties (properties that specialize a WPR in specific subtypes). 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 are the primary properties while encapsulation, lifetime dependency, transitivity, shareability, separability, mutability, and existential dependency are the secondary properties. The impact of secondary properties on composition is given in [2]:

- A component A is encapsulated into a component B when only B can access its services. This implies that A cannot be part of another component, and A do not have any relationship of any kind with any component outside of B.
- The property of Shareability 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 i.e. same WPR) or at a *global* level (a part shared among different kinds of wholes i.e. different WPRs). The reverse of local shareability is local exclusion while the reverse of global shareability is global exclusion.
- Separability allows a component to be separated from its whole. The mutability is the property that 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 implies inseparability and that separability implies mutability.
- There are nine cases of lifetime 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 co-incidence 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.
- 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 B interface.

2.2. SECONDARY PROPERTIES: RELATIONS AND COMBINATIONS

Certain properties have strong relationship between them. The existential dependency property is strongly dependent with the one of immutability/inseparability. In the 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 interactions make possible to define several subtypes of the WPR "Fig. 1,". For example, a relation for which the shareability, separability and mutability properties have been selected will characterize a WPR subtype called *aggregation*. Another relation possessing the non-shareability, encapsulation, existential dependency, immutability, inseparability and non-transitivity will characterize the subtype called *composition*.

The main benefit of this approach is the ability to add some precisely defined properties on the composition relation. It is possible in this approach to specify constraints on the composition itself. It makes possible to consider interactions between components during the modeling phase, early in the lifecycle.

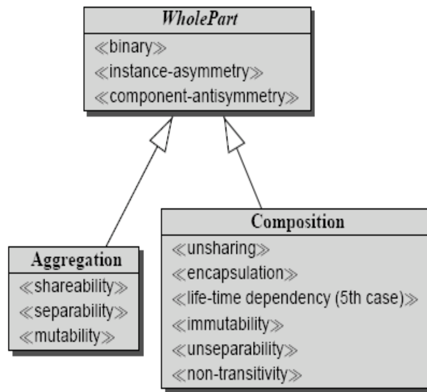

Figure-1 Whole-Part Subtypes

2.3. SOFTWARE CONNECTORS

Applications are made of a collection of reusable software entities (components) and mechanisms that permit their interaction (connectors). Connectors have many forms: On one hand, industrial approaches use simple connectors that are mainly point-to- point connections. On the other hand, academic approaches, like Architecture Description Languages (ADLs), recognizes complex connectors as first class design entities [9]. Hence, there is not a standard way to represent connectors [3].

A connector is a reification of an interaction, a communication or a coordination system. It is an autonomous entity at design level but neither deployable nor compilable. The connector is an architectural entity it must be connected in order to be deployed. It does not specify offered or required services but patterns of services. Hence, the interfaces are abstract and generic. They become more concrete later in the life cycle, on assembly with the components. This is done by adopting the interfaces type of the interacting components. The word plug is used to name the description of the connector ends. A plug is typed with a name and can have a multiplicity. At an abstract level, it exists to serve the communication and coordination needs of unspecified interacting components. Later in the life cycle, it describes all the interactions between special components by relying on the own interface specifications of these components. It also offers application independent interaction mechanisms like security and reliability.

A connector specifies the property it has to ensure, the number of different participants that are in position to interact, their status and their multiplicity. It is an architecture element that indicates the semantics of the communication and that evolves and changes in time to be more specified. It is designed, implemented and put on the shelf without any specified interfaces. The authors in [8] introduces a new vocabulary in order to designate each entity in its life cycle and has given a different name and a particular graphical representation to each entity in the life cycle with abstract parts (dotted lines) and concrete parts (solid lines). Connectors are represented by an ellipse, in order to differentiate them from components, and plugs are represented by circles around the ellipse "Fig. 2,". The grey parts show the abstract parts and the black parts shows the concrete parts of the connectors and of the different entities in its life cycle. The authors identify three entities that designate the transformations of the connector in its life cycle [8]:

An isolated connector is concretized as a family of off-the-shelf generators. These generators are in charge of implementing the connector abstraction over different communication protocols or platforms. The plugs are still abstract because the interacting components are not specified yet. When activating the generators, the plugs are destined to be transformed into proxies towards which the effective communication will take place.

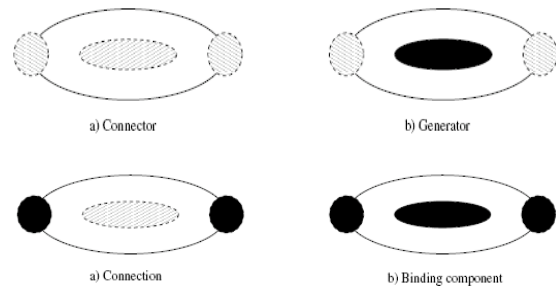

Figure-2 Connector Stages

At architecture level, the connector is linked with other components and forms all together the connection. The generic interfaces of the connector i.e. the plugs, become concrete by adopting the applicative interfaces (APIs) of the interacting components. They form the connection interfaces. Hence, the components are being connected without worrying about the underlying platform; only the semantic of the communication is needed. A connection is different from a connector. The connector exists alone. The connection appears when the components are specified, the connector is chosen and the unit is assembled.

Binding components are the result of the code generation. The connection interfaces, obtained by the connector and components assembly, are provided to the generator and fill its unspecified plugs. The generator generates the proxies that are to be deployed and form the binding components. They represent the realization of a communication property for the connected component's requirements (APIs) over the underlying system. Table 1 show the vocabulary used to distinguish different entities in the life cycle of a connector.

Table-1
Connector Vocabulary by Refinement and Abstraction

Level	Architecture (abstract)	Implementation (concrete)
Off-the-shelf	*Connector*	*Generators*
Assembled	*Connection*	*Binding components*

3. RELATIONSHIP BETWEEN THE TWO APPROACHES

3.1. LINKING THE TWO APPROACHES

A first contribution consists in matching the concepts of the two approaches as WPR properties can be applied to the software connector stages. This relationship is shown in Table 2.

Table-2
Relationship between WPR Properties and Software Connectors

Connectors	WPR Properties
Connector A	Shareability Seperability Mutability
Connector B	Unsharing Unseparability Immutability
Connector C	Encapsulation Non-Transitivity

All WPR properties have an impact on the connectors, the generators, the connections and the binding components, but at different abstract levels. The connector announces the property (the intent to use it); the generator implements the design choices that will be used to implement it; the connection specifies where the property applies; and the binding components actually implement the property. We can imagine connectors that are partially described, for instance if we do not have decided to accept transitivity or not. But it should eventually be decided so that the generator generates the binding components that ensure (or forbid) transitivity. Based on this notion we can have different types of connectors; each implementing different WPR properties. For example, we can have a connector called connector A, that implements Shareability, Separability and Mutability properties, another connector called Connector B that implements Unsharing, Unseparability and Immutability properties and Connector C that implements Encapsulation and Non-Transitivity properties. Table 2 shows different types of connectors based on the implementation of different WPR properties. After having defined these connectors, we can now move on to refine them by following the lifecycle of connector i.e. generator, connection and binding component. In this way each connector with WPR properties embedded in it will result in binding component, which is a full implementation of the connector.

3.2. TECHNICAL CONCLUSION AND LESSONS LEARNED

Software connectors embodied in itself the concept of software composition while the basic theme of the whole part relationship properties is to show how these properties can be used to specify the semantics of software composition [2]. Consequently, we can describe software connectors with the help of whole part relationship properties.

A true scenario would be to create a class diagram or a component diagram in UML and apply one of the WPR properties to one particular association in the model. We would then transform this model through the use of model transformation languages and then we will check the use of this property in the new model. If this property holds true in the new model, that would mean that by generating the code for this new model, we will be automatically inserting this property in software connectors. This would be a first step in validating our proposal. After this, we would like to find the relationships that exist between these properties when we apply them on software connectors. From this we would be able to deduce different variants for these properties and find out which properties are composable or orthogonal with each other. If we apply a property to a connector and we use this connector in another scenario and we check for the compatibility of this property in this new setup that is being done through model transformation languages that would be the prime success of our study.

We are using model transformation for the validation of our work, as in the source model we can apply WPR properties on complex associations and then in the target model, these properties will act as wrappers or adapters between the communicating components. On the basis of this new model, we will then generate the code with these properties Embedded in it.

For model transformation, we have different options of model transformation languages and tools e.g. Kermeta [10], ATL (Atlas Transformation Language) [11], TOPCASED [12].

4. CONCLUSION AND FUTURE WORK

WPR properties add some precisely-defined properties on the composition relation. It makes possible to consider interactions between

components during the modeling phase, early in the lifecycle. While looking into software connector, its definition and life cycle is used to implement complex communication abstraction. Indeed, to ensure properties like authentication, data compression, load balancing etc., current development methods mix them with the application code. By dedicating one connector to realizing such non-functional properties, thanks to the generation process, one offers the possibility to provide separation of concerns transparently to the application and independently of the platforms. In WPR, a precise description of the properties is given which is being undermined by the lack of implementation support. The software connector approach is a promising approach for the implementation of the connectors but it lacks the description of connectors. This paper is a contribution towards bridging the gap between the two approaches. The overall goal of this research work is to find a way for the conception of a Composition Modeling Language. An initial experiment will be to apply these properties to the different stages of software connectors.

REFERENCES

[1] Kung-Kiu Lau, Ling Ling and Perla Velasco Elizondo, "Towards Composing Software Components in both Design and Deployment Phases" The 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering Global Software Services and Architecture July 9 - 11, 2007**,** Tufts University, Medford (Boston area), Massachusetts, USA.

[2] Belloir N., Romeo F., Bruel J.-M., "Whole-part based composition approach: a case study" Euromicro Conference, 2004. Proceedings. 30th Volume, Issue, 31 Aug.-3 Sept. 2004 Page(s): 66 – 73 Digital Object Identifier 0.1109/EURMIC.2004.1333357.

[3] Selma Matougui and Antoine Beugnard, "How to Implement Software Connectors? A Reusable, Abstract and Adaptable Connector" L. Kutvonen and N. Alonstioti (Eds.): DAIS 2005, LNCS 3543, pp. 83–94, 2005. IFIP (International Federation for Information Processing 2005).

[4] V. Lakshmi Narasimhan, B. Hendradjaya, "Some theoretical considerations for a suite of metrics for the integration of software components" Inf. Sci. 177(3): 844-864 (2007).

[5] Pedro J. Clemente and Juan Hernández, "Aspect Component Based Software Engineering" this project has been financed by CICYT, project number TIC02-04309-C02-01.

[6] Szyperski C., *Component Software: Beyond Object-Oriented Programming* Addison-Wesley, 1998.

[7] Nicolas Belloir, Phd Thesis "Composition conceptuelle basée sur la relation Tout-Partie" LIUPPA (Laboratoire d'Informatique Université de Pau et des Pays de l'Adour, 2004).

[8] Selma Matougui and Antoine Beugnard, "Two Ways of Implementing Software Connections among Distributed Components" In International Symposium on Distributed Objects and Applications, DOA 2005, Agia Napa, Cyprus, Oct 31 - Nov 2 2005.

[9] M Shaw, "Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status" Tech. Report CS-94-107, Carnegie Mellon University Pittsburgh, PA, USA, 1994.

[10] Kermeta http://www.kermeta.org/

[11] Atlas Transformation Language http://www.eclipse.org/m2m/atl/

[12] TOPCASED http://topcased-mm.gforge.enseeiht.fr/website/index.html