Abstract

Reusability is a key factor for the success of the development of low-cost applications. Component-based software engineering (CBSE) aims to address this challenge by providing flexible and easy to use software components. Composability is a key concern of CBSE because software components collaborate with difficulty in spite of the fact that they are known to be compositional. In this paper we propose to formalize a special kind of composability, based on high coupling in which fine-grained components are encapsulated inside high-granularity components that are deployed on the same node. We specifically study the properties of the universal whole-part relationship within a composition framework. We constrain composition design by means of rules derived from the whole-part relationship. These constraints are used to generate contracts at implementation time. Finally, we incorporate built-in test functionality into components to support runtime validation.

1 Introduction

Developers of modern software systems need to be concerned with critical issues related to use of standards distribution of resources across network infrastructures, flexible adaptation to known and unknown (at development time) runtime environments, and reduced time-to-market windows. One approach to reducing the development costs involves reusing all or parts of existing code. Object technology claims to support such reuse. However, in spite of its considerable benefits, reusability can be difficult to achieve in an object oriented approach to development. Specifically, the use of inheritance as a reuse mechanism [1] is limited because the unit of reuse is a class. More promising approaches to reuse are design patterns [2] or components. Component-based technology is considered by some as enabling reuse that goes beyond inheritance [3].

Composability is considered to be an essential basis for component reuse. Moreover, based on the hypothesis that component specifications are critical to successful development, composability must be anticipated and explicitly addressed. Dependencies between components are often not formally specified, and, consequently, this makes intelligent assembly of behaviors difficult and sometimes not predictable [4].

We view composability from two angles. Horizontal composability refers to component binding and cooperation in distributed systems. Vertical composability means that whole (container) components process requests of client components. Part components are fully encapsulated (by whole components) and, above all, are not units of deployment in this particular context. Rather, they provide implementations for the components that they belong to: client requests can be delegated to part components. This approach offers flexibility through the introduction of the concept of a configuration interface [5] (also called diversity interface in [6]), that allows one to parameterize components in order to adapt them to different contexts.

This article proposes to extend the semantic properties of the whole-part relationship by adapting its formal base to software composition. Using previous results on Whole-Part properties in OO [7], we determine which properties apply to component composition and define these properties formally. 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. Finally we supply built-in test functionality to support composition checking.

After a discussion on the overall problem of composability in relation to the Whole-Part theory in section 2, we detail our proposal in section 3. We analyze the conditions under which part components can be encapsulated by whole components: encapsulation can be based on exclusiveness, emergent and resulting properties, and on links and constraints between Whole and Part component states and lifecycles. Before closing, we illustrate in section 4 how our approach may be supported in an existing Java-based library.
In this section, we present our proposal for a composition framework based on a theoretical framework for interpreting the Whole-Part Relationship (WPR in the remaining of this paper).

2.1 Composability

Component integration is sometimes viewed as a mechanism for connecting interfaces. However, such a simplistic view of integration is not sufficient to formalize composition. One must determine the properties of the assemblies through component compatibility at deployment time in particular [8].

**Composability** for a software component is the ability of the component to be systematically and easily combined with other components. This characteristic is of primary importance to allow the use and the re-use of a component. However, although “Components are for composition”, as noted by Szyperski [3], experience shows that software parts are not always components in the sense that they are not always intrinsically and directly composable. This observation is confirmed in [9]:

> However, this has led to design practices that assume components are “compositional” in all other behaviors: specially, that they can be integrated and still operate the same way in any environment. For many environments, this assumption does not hold.

Moreover, the increasing size of applications leads one to consider various levels of granularity and of composition during system development. This is for example the underlying assumption of the EDOC profile [10]:

> Having a rigorous and consistent way to understand and deal with the hierarchy of parts and compositions, how they work and interact at each level and how one level relates to the next, is absolutely necessary for achieve the business goals of a flexible and scalable information systems.

2.2 Whole-Components and Part-Components

The concept of Part-component was introduced in [11] through the *sub-component* name. In this paper, we use the term Part-component in order to conform to the EDOC terminology [10] which speaks about parts and compositions.

The general dependencies between Part-components and Whole-components appear in Fig. 1. Part-components realize the required interface of Whole-components. Both have configuration interfaces; the Whole-component interface is accessed by managers and Part-components are accessed by their owning components.

![Fig. 1 - Canonical organization between Whole-components and Part-components (UML formalism)](image-url)

2.3 WPR Features

Dependency variations between a Whole and its Parts may be expressed through the choice of a precise set of basic essential characteristics [7]. Some of them are always present in a WPR, 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). Table 1 summarizes this list of properties.

We discuss the relevance and the use of the secondary properties only in the context of software components combination. Secondary properties characterize several kinds of WPR. It is thus possible to specify different kinds of composition. A Part-component must be completely encapsulated by the Whole-component. Consequently, Part-component is not shareable in the global application architecture. Our approach supports granularity in the sense that a Part-component can itself be a Whole-component (composed of other Part-components) recursively.
Table 1. Whole-Part Relationship properties

The properties of encapsulation and non shareability have a direct effect on the property of transitivity of the WPR. The transitivity property can be stated as follow: A is related to B which is itself related to C, then A is related to C. For component systems, if A requires a service from C, this service will be requested through the intermediate component (B) and A will have no knowledge about the existence of C because of its encapsulation in B. In our strong coupling approach, the property of transitivity is thus not satisfied.

We believe that the principle of immutability must be satisfied by the Part-component. Indeed, this principle ensures that the Part-components are fixed in identity and in number. That guarantees the quality of service of the Whole-component while avoiding potential internal confusion.

We have formally demonstrated in [12] that immutability implies inseparability. Thus Part-components cannot be separated from their Whole-component.

We have identified nine candidate lifecycle dependencies (depending on whether Whole-Component is created before or after the Part-Component, or if they are simultaneous created. Only simultaneity (from a transactional point of view) of both lifecycles is eligible according to our characterization of composable. Indeed, computation required by clients require that components propagate requests to sub-components. Request processing can only occur if parts of components exist and are synchronized. Similarly, when a component is not used it can be deleted and consequently its parts are also deleted (parts are not shared).

3 Component Behavior

Our assumption is that the global state of a Whole-component is, recursively, the aggregation of the states of all its Part-components. Fig. 2 gives an example of a state-based representation of a Whole-component composed of a Part-component S0, which is itself composed of the Part-components S1, S2 and S3. This approach is justified by the fact that the Part-components are integral parts of their Whole-component and that their behavior cannot then be disjoint from the behavior of the Whole.

![Fig. 2 - State-based behavioral composition (UML notation)](image)

Linking the behaviors of wholes and parts is recognized as crucial. For example, in [13] a composition language, called Part-Whole Statecharts, that explicitly correlates the Statecharts of component entities to the state of some compound entities is described. If a component (A) has the same transition on its Statemachine found in another component (B), and both are parts of a compound entity (C), then one can include this event in the compound. So, there is no dependency between both components. This approach is valuable but has some limitations, especially for distributed applications. However, we are studying how this proposition can be used in our approach. For example, one of our concerns is about the relationship between the Part-Component A and the Part-Component B in Fig. 3.: how is it going to be translated in terms of interfaces and methods? The question is, if A proposes a service Sa, and B requires this service, could B directly use Sa or should the Whole-Component provide Sa to B?

![Fig. 3 – Is relationship between A and B is authorized?](image)
4 Implementation

We use a metamodelling approach to develop a theoretical framework for WPR. The idea is to add new metatypes and associated assertions within the existing UML metamodel in order to constrain the way composition occur.

4.1 The Metamodel

We propose to specialize the concept of component into Whole and Part components (see Fig. 4) plus a meta-aggregation between the two. We also specialize the concept of interface to obtain provided interface, required interface and management interface. The first two address client-server interaction issues. The third allows one to configure the component for adaptation purposes.

![Fig. 4 - An improved UML metamodel](image)

4.2 Definition of the properties constraints

All the available properties that can parameterize a WPR cannot be expressed at the metamodel level. There are at least two reasons for this. First, some properties are optional and including them at the metamodel level would be too restrictive. Second, the UML graphical notation does not allow the capture of all of the characteristics we need to express. The latter reason is a UML drawback that has been compensated via the introduction of the constraint language OCL (Object Constraint Language) [14].

In order to illustrate the use of OCL, we use it to express the primary property of asymmetry. Asymmetry of a relation means that this relation is non reflexive -- f(x, x) must be false -- and antisymmetrical -- f(x, y) implies !f(y, x). The corresponding OCL rule supplements the metamodel:

```
context WPRelationship inv Irreflexivity:
theWhole.instance->forAll(w | w.oclIsKindOf(Part)
    implies not w.part->includes(w))
thePart.instance->forAll(p | p.oclIsKindOf(Whole)
    implies not p.whole->includes(p))
```

The metamodelling approach allows us to express a number of WPR properties and thus provides a priori, structural, and behavioral, validation.

5 Validation via Built-In Test

In the previous section we have presented our approach to rigorously compose a priori components. In this section, we present how we a posteriori validate component compositions. We developed a technique called Built-In Test (BIT) that fosters the implementation of Whole and Part component dependencies as well as their coherent behaviors ruled by contracts. We have developed a Java library, called BIT/J, that allows us to address COTS product issues [15]. This library is based on the introspection capability of Java. With this technology, component users can manipulate components with a generic testing interface, in order to verify its behavior with respects to the specified constraints.
BIT is based on contract notion defined by Meyer [1]: each component must respect a contract with other components. This contract is specified during design in terms of pre, post-conditions and invariants. We propose to include into these contracts semantic constraints and composition properties to allow their verification.

We have developed an automatic generation tool that generates from a Java component the main part of the testing interface as well as a specific remote testing interface based on JMX [16] that allows remote control on a BIT component.

Fig 5 illustrates how to use the BIT/J library. The left side represents part of the library itself. Interfaces IBIT Query and State-based IBIT Query are generic testing interfaces. BIT state monitor allows one to implement into the BIT component the Statechart representing component behavior. Secondly, if the component is a COTS component it may not have a Statechart diagram. In this case a re-engineering process is needed in order to define a Statechart diagram. Our library also supports a non-state based approach. In this case, only a BIT component implementing IBIT Query interface is developed. BIT test case and State-based BIT test case allows one to define test cases that manipulate BIT component and generic interfaces. Our tool, from a Java component or a Java COTS component, automatically generates part of BIT component and of its corresponding tester (e.g. State-based BIT component tester). The tester implements JMX interfaces so it can be manipulated from a Web Browser.

6 Conclusion and future works

We have presented in this article an approach for the composition of components based on the theory of the whole-part relationship. It is based on a strong coupling mechanism. Its novelty relates to the specification of semantic constraints at the modeling level, through the concepts of Whole-component and Part-component. We have shown that this approach supports the prediction of the behavior of the assembly of components thanks to the strongly and semantically constrained composition relationship.

As a complementary validation approach, we allow the users of the provided components to check whether the acquired components match their expectation in terms
of services and behavior. In order to do so we provide a set of testing and management functionality in the components. The BIT/J library we have developed within the context of the COMPONENT+ project\(^1\) allows us to provide such facilities even for Java-based COTS components.

We are currently integrating our approach within a composition framework using a UML tool supporting the definition of metamodels and the integration of specific modules. This additional module will implement our approach through a set of modeling and checking functionality. The tool will then support model analysis as well as the generation of code that implements the properties specified at modeling time and, in particular, the intrinsic properties to the WPR. Finally, we plan to extend our approach to take into account the specific problems related to the use of distributed components.

7 References


\(^1\) European IST-1999-20162 project. See [http://www.component-plus.org](http://www.component-plus.org) for more details.