UML 2.0 Components and Fractal: An Analysis

Vladimir Mencl[1,2], Matej Polak[2]

[1]*United Nations University International Institute for Software Technology*
UNU-IIST, P.O. Box 3058, Macao, mencl@iist.unu.edu, http://www.iist.unu.edu/
[2]*Charles University, Faculty of Mathematics and Physics, Department of Software Engineering*
Malostranske namesti 25, 118 00 Prague, Czech Republic
{*mencl,polak*}*@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz/*

1. Introduction

The newly emerged standard UML 2.0 [8] provides a framework for modeling software components. In its design, the existing industrial component models (EJB, CCM, COM+, and .NET) have been explicitly considered. However, these models do not possess the features present in advanced research component models, in particular hierarchical composition. Yet, the rich set of modeling constructs available in UML 2.0 may be sufficient even for these component models.

In this paper, we analyze the component modeling framework provided by UML 2.0 with respect to how its abstractions match the concepts used in currently available advanced component models. Besides a general analysis, a closely related goal is to explore in detail how UML 2.0 can be used to model applications in the selected component models Fractal [1] and SOFA [4].

In Sect. 2, we briefly introduce the relevant key features of UML 2.0. We follow by Sect. 3 where we analyze how UML 2.0 can be used for advanced component models, and we propose a mapping for the Fractal component model in Sect. 4.

2. UML 2.0 Components: Key Features

Structuring a Classifier. UML 2.0 introduces the Internal Structures framework, which is used to capture the internal structure of a component (and can be also used for hierarchical components). The metaclass StructuredClassifier allows to decompose the functionality of a Classifier into several *parts*. A part is a Property of the owning StructuredClassifier owned via the parts association; technically, the type attribute of a part specifies the type of the classifier that will be instantiated within an instance of the owning structured classifier. Further, parts may be interconnected via connectors, which correspond to future links to be established among the corresponding instances.

Encapsulating communication of a Classifier. In the Composite Structures framework, a Port may be used to explicitly capture the external communication of a classifier. An EncapsulatedClassifier may be associated with zero or more ports, and a Port is associated with a set of provided and required interfaces.

UML 2.0 Components. The concepts of internal structure and encapsulation of communication are merged in the metaclass Component, the key construct for component modeling. The metaclass Component also inherits from the metaclass Class (as redefined in the Structured Classes package) and gains the ability to have methods and attributes and to participate in associations and generalizations. Further, besides the associated ports, a Component may be also directly associated with a set of provided and required interfaces.

In the following section, we analyze how these UML 2.0 constructs can be used to model components in advanced component models, what options are available, and what restrictions are associated.

3. UML 2.0 Components: Analysis

Provisions and Requirements. In most component models, a component features a collection of provided and required interfaces, identified by their type and often also a name to distinguish multiple occurrences, or "instances", of the same interface type.

In the case of interfaces directly associated with a Component, UML 2.0 provides no way to specify the name of an interface "instance". This technique can therefore be only used to specify anonymous provisions and requirements, specified only by the interface type.

When ports are used to associate interfaces with a component, a name may be assigned to a port, specifying a named set of provisions and requirements of the component. However, as only a single name may be specified for a port, if a name is required for each interface (such as in Fractal and SOFA), the only solution is

to use solely one interface per port. Further, as a Port may have a multiplicity specified, this approach may be used to model multiple (or "collection") interfaces featured in some component models (e.g., Fractal).

Subcomponents. UML 2.0 supports two ways of modeling subcomponents; besides the internal structures framework, it is also possible to use the Namespace feature of the Component metaclass. This permits a component to own model elements, in particular instances of the metaclasses Component, InstanceSpecification, Class, and Interface.

In hierarchical component models, subcomponents are usually specified as instances of component types, where not only the types, but also the subcomponent instances are identified by a name. Property and InstanceSpecification are both a NamedElement, and so is the type definition element they reference. Therefore, both the approaches considered allow to specify both an instance name and a type name.

Subcomponents as parts. When specifying a subcomponent as a part, the type attribute of the Property representing the part is used to refer to the subcomponent definition. The advantage of this approach is that for each part, its multiplicity may be specified, allowing to model a collection of subcomponents.

Subcomponents as nested elements. Subcomponents may also be modeled by nesting a Component definition inside the Component element of the owning component. However, such a construct defines only a component type and not an instance, and must be combined with an InstanceSpecification model element, representing the desired subcomponent instances. An InstanceSpecification may of course also refer to types defined outside of its owning component.

Note that UML 2.0 leaves up to an implementation to define the syntax and semantics of the specification attribute of an InstanceSpecification. However, for the component models concerned, it is possible to introduce both expressions creating new component instances as well as expressions sharing component instances defined elsewhere.

The key advantage of a nested component definition is the compact definition, without the need to first define the component type elsewhere. The use of an InstanceSpecification allows to model shared components. However, contrary to subcomponents as parts, this approach does not permit to model multiplicities of subcomponents.

We demonstrate both the possible approaches in Fig. 1, where the list subcomponent of Wnd is modeled as a part (employing a multiplicity specification), MainWnd is an embedded component type definition, and main is an InstanceSpecification.

Connectors. A Connector in UML 2.0 specifies a link along which two instances will be able to communicate. For components, the role of connectors is to connect their provisions and requirements. For a UML 2.0 component, these may be represented either as ports featuring provided and required interfaces, or as interfaces directly associated with the component.

In UML 2.0, a Connector may connect any two model elements that are a ConnectableElement; this includes a Property, and therefore a part representing a subcomponent, and a Port. The UML 2.0 specification however has certain deficiencies and inconsistencies in the connectors framework. The UML 2.0 Finalization Task Force (FTF) report [9] issues 7247, 7248, 7249 and 7251 all identify that an Interface is not a ConnectableElement, while the Connector specification (sect. *Constraints*) assumes that it may connect interfaces. The reported issues have been acknowledged by the FTF, but a solution was not provided; instead, the issues were deferred for the next Revision Task Force (RTF). Hence, the connectors may only link ports, connecting all of their matching interfaces at once. Alternatively, a Connector may also be attached to a subcomponent specified as a part, attaching all of its interfaces. It is however not possible to selectively connect individual interfaces of a port.

Note that in the case of subcomponents specified as parts, the connector is attached to the Port of the component typing the part, and the partWithPort association of each of its ConnectorEnd elements would link to the respective part. However, UML 2.0 does not explicitly address the case of an InstanceSpecification, and it is not possible to attach a connector to the subcomponent (an InstanceSpecification is not a ConnectableElement). If the Connector is attached to a Port of the InstanceSpecification, there are no means to relate the Connector to the specific InstanceSpecification (as it is done for a part with the partWithPort association). Note that in existing tools implementing UML 2.0, proprietary alternations of UML 2.0 are used to circumvent this problem — as we also show in Fig. 1, where InstanceSpecification main features ports connected to the the components Wnd and list.

UML 2.0 distinguishes two kinds of connectors: *assembly* connectors to connect provisions and requirements of peer (sub-)components, and *delegate* connectors to link provisions (or requirements) of a component to provisions (or requirements) of one of its subcomponents. These modeling constructs are sufficient for the needs of the component models we consider, even though certain issues may have to be resolved in the mapping of UML 2.0 to such a model, as some component models distinguish in the case of delega-
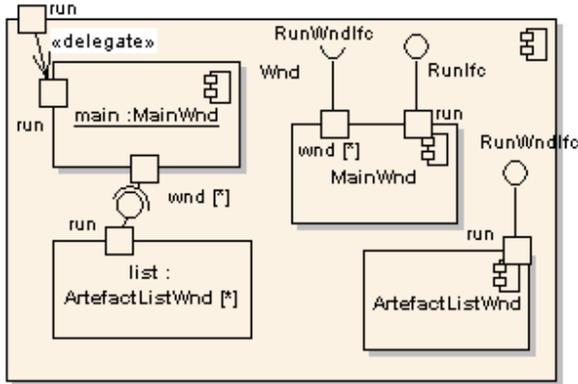
Figure 1. Fragment of component model of a document management system GUI. Component `Wnd` contains definitions of `MainWnd` and `ArtefactListWnd`, instance `main` and part `list`.

tion connectors between actual delegation (provided-to-provided) and subsumption (required-to-required).

Attributes and Methods. As the Component metaclass inherits from Class, it may feature attributes and methods. Attributes may be used to model component configuration parameters (also called attributes or properties in some component models). Method declarations (abstract methods) owned by a component may model provided operations specified directly in the component contract, instead of being explicitly included in an interface. (e.g., in Corba CCM).

Realization. A primitive component may directly implement its provided interfaces and their operations in its owned methods. Alternatively, the provisions may be implemented by a *realizing classifier* (usually a Class), linked to the component via the Realization , a specialized Dependency relationship.

Inheritance. As a Component is technically a specialized Class, it may participate in Generalization (the UML relation modeling inheritance). By inheriting from a Class, a component obtains all its attributes and methods, which may implement the functionality provided by the component. Further, a generalization relation to a parent component allows to model *component inheritance*, where a Component in addition acquires all provisions and requirements of its parent component, as well as its internal structure.

4. Modeling Fractal Components

Having analyzed the UML 2.0 component modeling capabilities in general, we now propose how UML 2.0 can be employed to model components for the Fractal component model [1]. The notion of a component

in UML 2.0 and Fractal are very close; nevertheless, we introduce the stereotype `FractalComponent` to identify Fractal components and to store additional information needed in the *tagged values* it defines.

Note that the Fractal specification leaves semantics of certain concepts, namely interface signatures and content descriptors, open; it is to be determined by either the target platform or the concrete Fractal implementation. In the mapping, we consider Java as the target platform; we thus assume the interface signatures and content descriptors are both specified as a fully qualified Java class name.

Ports and Interfaces. While UML 2.0 components may specify provisions and requirements as either direct interfaces or ports, Fractal components only have interfaces. Also, Fractal interfaces are specified by a name, signature, and role (either client or server), and may also have cardinality and contingency. However, in UML, interfaces do not have names, only ports do.

In our mapping, we consider both ways of specifying component interfaces in UML 2.0. As interfaces attached directly to a Component do not have a name (and Fractal requires a unique name for each interface), the mapping dynamically assigns a unique name to each such interface. Further, as in this case there is no way to specify multiplicity, the corresponding Fractal interface is always a mandatory interface with single cardinality.

In the case of ports, we allow only a single interface to be associated with a Port; the name of the port becomes the name of the Fractal interface. If the port has a multiplicity higher than one, the resulting Fractal interface is a collection interface. If the lower bound of the port's multiplicity is less than one, the contingency of the interface is optional.

Subcomponents. A specific feature of Fractal ADL is embedding a subcomponent definition in the definition of its parent component. While the UML 2.0 nested component definition appears to be its equivalent, it only defines a component type and must be accompanied by an InstanceSpecification. Such a pair is together mapped to Fractal embedded subcomponent definition. A subcomponent specified as either a part or an InstanceSpecification is mapped to a Fractal subcomponent created as an instance of an externally defined component type. An InstanceSpecification with a path expression to a component instance (in the Fractal ADL syntax) is mapped to a shared component.

Connectors. The connectors specified in a component definition are mapped to Fractal bindings: assembly connectors to bindings between interfaces of subcomponents (of different role, i.e., client or server) and delegate connectors between interfaces of the containing component and a subcomponent (of the same role).

Attributes. The attributes of a UML Component are mapped to attributes of the corresponding Fractal component; the AttributeController interface is generated as a part of the mapping. As Fractal permits only attributes of primitive types, the attribute types in the UML model have also to be restricted to only the primitive types mappable to the target language (Java).

Inheritance. UML 2.0 components feature multiple inheritance, and so does Fractal ADL. Hence, we map each inheritance relationship of a UML Component to another Component into an inheritance relationship among the corresponding Fractal components. A Component may also inherit from a Class or an Interface; we permit these relationships only for a primitive component, and map them to corresponding relationships of the component's content class. Due to the Java platform restrictions, we permit only single inheritance from a Class (but multiple from an Interface).

Realization. A primitive component may have a Realization relationship to a Class; the class becomes the content class of the corresponding Fractal component. If no class is associated with the component, a skeleton of the content class may be generated. To control this feature, the `FractalComponent` stereotype introduces a (boolean) tagged value `FractalGenerateContentClass`.

Behavior Specifications. As the only way to specify behavior of Fractal components have so far been behavior protocols [3, 5], we introduced the tagged value `BehaviorProtocol` in the `FractalComponent` stereotype. However, to align this mapping with the recent coordinated efforts on specifying behavior of Fractal components, we intend to also consider the Behavior owned by the Component. In the specific case of OpaqueBehavior, its attributes body and language can be straightforwardly mapped to the corresponding attributes of the recently proposed Fractal ADL `behavior` element.

Additional Component Models: SOFA. We also developed a mapping for the SOFA component model; however space constraint do not permit us to elaborate on the mapping here. A key difference is that as SOFA considers two levels of component definitions, *frame* and *architecture*, two different Component stereotypes had to be introduced, `SOFAFrame` and `SOFAArchitecture`.

5. Evaluation & Conclusion

Although several projects already support visual modeling of SOFA and Fractal components [2, 7], none of them so far is based on an existing standardized modeling language [8]. The need for UML 2.0 mappings for concrete component models is widely accepted; OMG currently develops a UML 2.0 Profile for the CORBA Component Model [10] (also employing stereotypes to denote CORBA components).

We have implemented the proposed Fractal mapping of UML 2.0 in a plugin for the Enterprise Architect UML tool. Besides checking various constraints documented in [6], the plugin also generates all the relevant code artifacts: the Fractal ADL definition of the component, the Java interfaces and the AttributeController interface (if needed), and also the skeleton of the content class of primitive components (with implementations of the AttributeController and BindingController methods, and skeletons of the methods offered in the component's provided interfaces).

As an aside, our analysis of UML 2.0 finds that its Composite Structures and Components frameworks are heavily underspecified, and a user of the specification (often the tool developer) is forced to find his/her own interpretation of the specification, with the inherent risks on model interchangeability.

Our current work is focused on proposing a concise set of amendments to the specification, which would resolve the most crucial ambiguities and deficiencies.

References

[1] Bruneton, E., Coupaye, T., Leclerc, M., Quema, V., Stefani, J-B.: *An Open Component Model and Its Support in Java*, Proceedings of CBSE 2004, May 24-25, 2004, Edinburgh, UK, LNCS 3054, Springer, 2004

[2] Cifka, M.: *Visual Development of Software Components*, Master Thesis, advisor: Petr Tuma, Charles University, Prague, Sep 2002

[3] Component Reliability Extensions for Fractal Component Model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`

[4] Plasil, F., Balek, D., Janecek, R.: *SOFA/DCUP Architecture for Component Trading and Dynamic Updating*, In Proceedings of ICCDS '98, Annapolis, IEEE CS, 1998

[5] Plasil, F., Visnovsky, S.: *Behavior Protocols forSoftware Components*, IEEE Trans. Software Eng. 28(11), 2002

[6] Polak, M.: *UML 2.0 Components*, Master Thesis, advisor: Vladimir Mencl, Charles Univ., Prague, Sep 2005

[7] ObjectWeb: *Fractal Explorer*, `http://fractal.objectweb.org/tutorials/explorer/`

[8] OMG: *Unified Modeling Language: Superstructure*, version 2.0, Final Adopted Specification, formal/05-07-04, `http://www.omg.org/uml/`

[9] OMG: *FTF Report of the UML 2.0 Superstructure Finalization Task Force*, ptc/04-10-01, `http://www.omg.org/uml/`

[10] OMG: *UML Profile for CORBA and CORBA Components*, Initial Submission, mars/2005-11-05, `http://www.omg.org/uml/`