A Close Look at Composition Languages

Florian Heidenreich Jendrik Johannes
Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden, Germany
{Florian.Heidenreich,Jendrik.Johannes,Uwe.Assmann}
@tu-dresden.de

Steffen Zschaler
University of Lancaster
Computer Science Department
Lancaster, United Kingdom
szschaler@acm.org

Abstract

A large number of different composition systems and techniques have been developed over the last years. To compare their relative benefits and drawbacks, we need a common vocabulary for describing elements of composition systems. This paper contributes to the search for such a vocabulary by taking a closer look at the structure of composition languages—that is, languages used for describing compositions—based on a survey of eight different composition systems.

Categories and Subject Descriptors A.1 [*Introductory and Survey*]; D.2.11 [*Software Architectures*]: Languages, Patterns

General Terms Design, Theory

Keywords Elements of Composition Systems, Composition Languages, Terminology

1. Introduction

Decomposition and modularisation have been used successfully to deal with complex system development. Whenever we decompose a system design or implementation, we also need ways of putting the parts together again. Typically, this is achieved by using so-called composition systems. Over time, a large number of composition systems and approaches have been developed. To be able to compare these, we need a common vocabulary describing salient elements of composition systems. This paper contributes to the ongoing search for such common vocabulary.

Composition systems and their elements have been studied for some time. (Medvidovic and Taylor 2000), were the first to discuss the elements of a composed system. They positioned their discussion in the context of Architecture Description Languages (ADLs) and found components, connectors and configurations as the central elements of an ADL. Later, (Aßmann 2003) introduced a more general characterisation of composition systems by distinguishing the elements of component model, composition technique, and composition language. In this paper, we focus on the internal structure of a composition language and define a common vocabulary for elements of a composition language; to the best of our knowledge the first attempt in this direction.

The remainder of this paper is structured as follows: In the next section, we propose new vocabulary for analysing composition descriptions. To evaluate whether this vocabulary is sufficiently general for a diverse set of composition systems, Sect. 3 then applies it to eight specific composition systems. Finally, Sect. 4 concludes the paper.

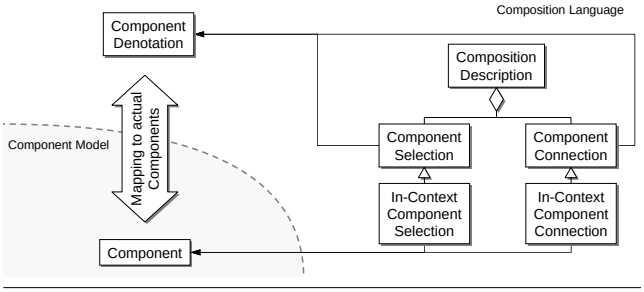

Figure 1. Elements of composition languages

2. Elements of Composition Languages

Figure 1 shows the elements of a composition language as we see them. In the lower left corner are the actual components. We do not care about the specific form in which these components exist: they could be source or binary components, or even, for example, just logical subdivisions of a large monolithic model.

For each of these components, we have a denotation in our composition language. Typically, we use these denotations to express compositions. Component denotations can take many forms: They can be simple names that stand for components, but they can also be symbols that need to be mapped to components in more complex ways—for example, we could use `#ifdef` symbols to denote components in the configuration of a C++ program. To handle this wide range of possible component denotations, a mapping between the denotations and the actual components is required. If denotations are only names, this mapping is simple. However, it can become arbitrarily complex in other cases.

Compositions are described in two steps: First, we need to select the components to be composed. Then, we need to describe how these components are to be connected. Depending on the composition system, both steps will be expressed implicitly on the component level or explicitly on the level of component denotations.

In the following, we define these concepts in more detail.

2.1 Component

We use the term *component* to refer to artefacts or groups of artefacts that will eventually form part of the composed system. Usually, one (ideally reusable) concern is contained in one component. Components need not be clearly identifiable physical entities: A single file may form a component, but a set of related model elements from different models may also form one component. Which information is considered as a component depends on the concrete composition system in use. Components can have hierarchical structure; that is, one component may contain other components.[1]

2.2 Component Denotation

Component selections and connections are often expressed using abstract representations of the actual component. We use the term *component denotation* to refer to such abstract representations. Component denotations can be simple names that stand for individual components, but they can also be complex structures representing components and certain selected properties of these components.

2.3 Mapping from Denotation to Component

The meaning of component denotations is defined by mapping them to actual components. Such mappings can be simple mappings that associate a symbol in the component denotation with a single-artefact component or they can be complex mappings that associate a single component-denotation symbol with a group of artefacts, possibly from different physical containers (e.g., files). Mappings can be defined extensionally by enumerating all denotational elements and their associated components or intentionally through rules that compute the components associated with a denotational element.

2.4 Component Selection

To describe how a system is composed from component, we must first select the components to be used. We use the term *component selection* to refer to this step and, in particular, to the part of a composition program expressing this. Component Selection normally uses component denotations to reference the components to be selected. A special kind of component selection—in-context component selection—is inserted directly in the definition of one component and implicitly selects this component. It uses component denotations to select additional components, though.

2.5 Component Connection

After selecting the components to be composed, we need to specify the connection between them. We call this step *component connection* and, also use this term for the part of a composition program expressing this. Normally, component connections use component denotations to reference all components to be connected. A special kind of component connection—in-context component connection—can be used directly inside the definition of one component and automatically references this component and specific implicit connection points within it. It uses component denotations to reference other components to be connected.

3. A Survey of Composition Systems

In this section, we survey a number of composition systems and show how they fit with the vocabulary we introduced in the previous section.

This survey serves two purposes: 1) to explain the individual terms of our vocabulary by providing concrete examples, and 2) to evaluate our vocabulary by showing that it can be applied to a sufficiently large set of composition systems.

3.1 Black-Box Composition

Classical, black-box component systems, such as CORBA (Object Management Group 2008) or Enterprise Java Beans (EJBs) (Microsystems 2001), have been available for some time. They typically provide an infrastructure that manages binary-format components and establishes connections and communications between them at runtime. One important part of this infrastructure is a naming service that allows components to be identified through some developer-chosen name. Components connect with each other by referencing these names in their source code or in so-called deployment descriptors.

Components come in the form of binary files. Depending on the specific component systems, these are machine-code binaries such as executables or dynamically linked libraries in the case of CORBA or byte-code binaries as in the case of EJB.

[1] The definition is generic on purpose: We focus on composition *languages* and are not overly interested in the details of what constitutes a component.

Component Denotation is done using textual identifiers. Some component systems support hierarchical name spaces and structured names. If so-called trading services are used, components can also be denoted by logical expressions over their properties.

Mapping Standard mapping is performed in the naming service. This mapping is explicitly defined by registering individual components with the naming service under a specific name. In the case of trading services, the mapping is more complicated and involves some amount of reasoning and matchmaking over the divers logical description terms. Still, however, the mapping is explicit insofar as each component is explicitly associated with a number of logical description terms.

Component Selection happens explicitly by naming associated components either in calls to the naming service or in so-called deployment descriptors that are delivered with a component. In both cases, all associated components are mentioned in the context of the component using them. Therefore, component selection is in-context.

Component Connection is combined with component selection. Therefore, it also happens in-context.

3.2 Aspect-Oriented Programming

In Aspect-Oriented Programming (AOP) (Kiczales et al. 1997), pieces of code—advices—are distributed over a core set of modules in a weaving process. Aspect definitions use pointcut specifications (that is, specifications of sets of joinpoints—points in the execution of the core modules) to specify where advice code should be woven. What joinpoints are available depends on the execution model of the programming language of the core modules and on the specific aspect approach.

Components are the core modules and the individual pieces of advice.

Component Denotation Advice components are denoted by simple names that developers associate with an aspect definition. Core-module components are denoted through a rich set of pointcut specifications referring to individual points in the execution of the core modules.

Mapping of advice denotations is implicit: A name used is mapped to all the advice of an aspect definition of the same name. Mapping core modules is also implicit: How pointcut expressions are mapped to specific points in the execution is hard-coded in the aspect weaver.

Component Selection Advice components are selected explicitly by making them available to the aspect weaver. Core module components are selected explicitly through pointcut expressions.

Component Connection is defined explicitly by associating a particular piece of advice with a particular pointcut expression. It is in-context, because pointcuts are expressed directly in the context of an aspect.

3.3 Model Weaving

Model Weaving often refers to linking two or more models by way of a weaving model (AMW Project Team 2008a) or link model (Kolovos et al. 2008). A weaving model contains a set of weaving links that link two or more model elements.

While model weaving can be applied for many model management tasks, it can in particular be used for expressing different kinds of model compositions. We look at two applications of model weaving for composition. In (AMW Project Team 2008b), a weaving model links elements of models that should be combined during composition. In (AMW Project Team 2008c), elements of metamodels are linked from which concrete compositions of models (instances of the linked metamodels) are derived. Note that there might well be other applications of model weaving where composition languages can be found and classified.

Components are models; that is, instances of a metamodel.

Component Denotation Components are denoted by IDs or path expressions that identify single elements in the models (AMW Project Team 2008b) or metamodels respectively (AMW Project Team 2008c).

Mapping is done implicitly: in (AMW Project Team 2008b) it is resolving an ID or expression to a model element. In (AMW Project Team 2008c) it uses the instance-of relationship.

Component Selection is selecting models or model elements to participate in the composition. In (AMW Project Team 2008b), single model elements are selected during the definition of a weaving model. In (AMW Project Team 2008c), metamodel elements are selected during the definition of a weaving model.

Component Connection In (AMW Project Team 2008b), component connections are explicitly defined through the definition of weaving links in the weaving model between the prior selected model elements. In (AMW Project Team 2008c), the component connection is defined on the meta-level.

3.4 Invasive Software Composition

Invasive Software Composition (ISC) (Aßmann 2003; Henriksson et al. 2008) is a language-independent software composition formalism. It defines a basic component model and basic composition operators independent of concrete languages and composition systems. On top of this, composition systems (including composition languages) can be built for arbitrary languages. In our tool Reuseware (Reuseware Project Team; Heidenreich et al. 2008a), we implemented the concepts of ISC for grammar-based and metamodel-based languages and provided a development environment

for composition systems based on ISC. Here we classify this implementation.

A fundamental principle of ISC is that each component—a fragment—has an explicit composition interface through which it can be addressed for composition. In Reuseware, fragments are either models or programs (i.e., instances of context-free grammars). The composition interface exposes selected model or program elements to be accessed during composition. Such elements are distinguished into reference points (can be accessed or extracted) and variation points (can be replaced). Concrete compositions are defined by composition links, linking reference and variation points in composition programs. Composition programs can be executed by statically replacing variation with reference points.

A composition system for a concrete language in Reuseware is created by defining how composition interfaces can be defined for or derived from models or programs written in that language. This definition contains rules that classify elements in models as reference or variation points. Furthermore, Reuseware allows the injection of constructs from Reuseware's generic composition language into a language for which a composition system is defined. This enables definition of composition programs inside of fragments.

Components A model is a component in Reuseware, if a composition system has been defined for the model's metamodel. A program is a component in Reuseware, if a composition system has been defined for the context-free grammar of the corresponding programming language.

Component Denotation Components are denoted by their name and by composition interfaces consisting of reference and variation points.

Mapping is implicit for a concrete composition system. It is, however, meta-explicit since it is defined and can be modified for each composition system separately in Reuseware.

Component Selection happens explicitly by selecting models or programs. It can be done in an external composition program, but also in-context, if the defined composition system supports it.

Component Connection is explicitly defined in form of composition links. This can be done in an external composition program, but also in-context, if the defined composition system supports it.

3.5 Template-Based Code Generation

Templates are documents that contain template parameters instead of concrete data at certain positions. Usually, templates are just regarded as plain text and do not have to conform to any specific language. Therefore, templates can be defined for any kind of digital document (Java classes, HTML pages, configuration files, etc.) and processed by a template engine like JET, StringTemplate, JSP, Velocity or MOFScript. Those engines take templates and data as input and compose them into complete documents. Since all mentioned engines work in a similar fashion, the classification below applies to all of them.

Components are: 1) Structured data (e.g., XML files, Java objects, models) and 2) templates. The space of available components is typically determined in a surrounding program by selecting a set of templates and associating data with named parameters in the template engine.

Component Denotation exists in two forms: templates are denoted through a naming scheme specific to the template engine. Data is denoted through template parameters, which are names sometimes associated with type information.

Mapping happens in two steps: 1) Components are explicitly associated with names by a) providing named templates and b) associating structured data with named parameters in the configuration of a template engine. 2) The template has an implicit internal mechanism for deriving denotations from these names and the components themselves. For example, some template engines will make available all elements of a data structure that are accessible through operations named `getXXX()`. However, these components will be made available under a name that excludes the `get`.

Component Selection is performed explicitly in the context of the templates.

Component Connection is done explicitly in-context at the same position where the selection takes place. Selection and connection are coupled.

3.6 Feature-Oriented Programming

In Feature-Oriented Programming (FOP) a *feature* is seen as an increment in program development and functionality. It can be used in Software Product Line Engineering (SPLE) (Pohl et al. 2005) to define and synthesise programs based on a unique composition of features. One system to define and execute such compositions is AHEAD[2] (Batory et al. 2004), where each feature is a nested tuple of unary functions (also called *deltas*). An example of such functions are Jak files, which add a certain increment in functionality to an existing Java class that shares the same name:

```
feature EnergySaving;

refines class MusicPlayer {
  autoSuspend() {...}
}
```

Components are features contained in files. The features are expressed in a language that can be composed by the AHEAD tool suite. For example, Jak components are

[2] Algebraic Hierarchical Equations for Application Design (AHEAD)

Java classes extended with domain-specific notions for defining refinements.

Component Denotation Components are denoted by the names of the files they are contained in.

Mapping Within AHEAD, features are mapped implicitly to their concrete realisation by using their names.

Component Selection happens explicitly by referencing specific features using their names in an AHEAD composition program.

Component Connection is usually done in-context of the feature module by using the denotation of the base component that is subject for refinement.

3.7 Feature-Driven Product Derivation

Variability modelling is used to express common and variable parts within Software Product Line Engineering (SPLE) and to explicitly define constraints between variable parts—features. Variability modelling abstracts from concrete feature realisation through feature models which is a powerful notion to handle the increased complexity in SPLE (Czarnecki 2005; Kang et al. 1990).

However, to build concrete products from a product line, features have to be realised using software artefacts shared across the product line. While variability modelling resides in the problem space, the realisation of features is part of the solution space. To instantiate products from a product line, feature realisations in the solution space must be included according to the presence of the features in a variant model; that is, a concrete selection of features from a feature model.

To support this transition from problem space to solution space in an automated way, a mapping from features to software artefacts that realise the features is needed. As an example, our tool FeatureMapper (Heidenreich et al. 2008b) allows for both defining and interpreting such mappings in a non-invasive way, that is, without changing the software artefacts.

Components are artefacts in models, that is, elements from models.

Component Denotation Components are denoted using names in the feature models. An important property is the ability to define constraints between components in the feature model.

Mapping Within the FeatureMapper, denotations—features from feature models—are explicitly mapped to concrete realisation components by the use of a mapping model.

Component Selection happens by selecting specific features from a feature model to build a concrete variant of the product line.

Component Connection In its default instantiation, our mapping framework works on components that are connected in solution models. During interpretation of the mapping model, solution artefacts are preserved and removed from the solution models depending on presence or absence of the corresponding features in a variant model. That is, we use a special form of in-context connection where every component is referenced directly.

3.8 Using IfDef Statements with the C Preprocessor

The C preprocessor (CPP) allows the definition and evaluation of `#define` constants. These can be used for many purposes in writing programs and managing their configurations. One very typical use is exemplified in the code snippet below:

```
#define USE_ENERGY_SAVING 1
...
#ifdef USE_ENERGY_SAVING
  // Component that saves energy
  cout << "Switched off your music player?"
       << endl;
#else
  // Component that wastes energy
  cout << "Want to start another device?"
       << endl;
#endif
```

It can be seen that this use of `#define` is actually quite similar to feature-driven development as discussed above: Here also, components are connected in-context, but selected through their denotations which are given as the labels of `#define` constants. Hence, this use of `#define` creates a composition system in our terminology:

Components are pieces of source code surrounded by `#ifdef ...#else` or `#ifdef ...#endif`.

Component Denotation is done by using `#define` constant labels.

Mapping The mapping of these labels onto components is given explicitly through `#ifdef ...#else ...#endif` structures in the source code.

Component Selection happens explicitly by defining specific `#define` constants either in some piece of source code or as explicit command-line parameters to the preprocessor.

Component Connection is done completely in-context, given by the order in which components are arranged within source-code files.

4. Conclusions

In this paper, we have proposed new terminology for describing the structure of composition languages. We have shown how this vocabulary can be applied to a large and diverse collection of current composition systems. Such vocabulary is an important prerequisite for comparative studies of different composition systems and composition languages.

In applying our proposed vocabulary to a number of composition systems, we have found—beyond showing that it is sufficiently general to cover a broad range of composition systems—two important factors for distinguishing between composition systems:

1. *Richness of component denotations.* Component denotations used in different composition systems range from simple names that stand for specific components to complex structures providing additional information about a component and its interface (e.g., feature models or component interfaces in invasive software composition). The more complex a component denotation, the more precisely can composition programs be expressed. Less complex denotations, on the other hand, are much easier to use for describing compositions.
2. *Definition of the mapping between denotations and components.* We have found that different composition systems use different techniques for defining a mapping between denotations and actual components. There are three possibilities:
   (a) *Explicit mappings,* where users of the composition language explicitly relate denotations and components in composition programs (cf. Sects. 3.1, 3.5, 3.7, 3.8);
   (b) *Implicit mappings,* where the relation is implicitly defined in the composition system (cf. Sects. 3.2, 3.3, 3.5, 3.6); and
   (c) *Meta-explicit mappings,* where the relation between denotations and components is provided explicitly, but not individually for each composition program, but rather as a set of rules that can be applied to many different composition programs (cf. Sect. 3.4).

Based on these findings, it is now an interesting question to study how these design decisions influence the usability of composition languages. What are good criteria for selecting one or the other type of denotation or mapping strategy for a specific composition language or even project?

Acknowledgments

This research has been co-funded by the European Commission within the FP6 project MODELPLEX contract number 034081 and by the German Ministry of Education and Research (BMBF) within the project feasiPLe.

References

AMW Project Team. Atlas model weaver, 2008a. URL `http://eclipse.org/gmt/amw/`. Last accessed August 08, 2008.

AMW Project Team. Amw use case – aspect oriented modeling, 2008b. URL `http://www.eclipse.org/gmt/amw/usecases/AOM/`. Last accessed August 08, 2008.

AMW Project Team. Amw use case – merge of geographical data (gml) with election statistics into svg, 2008c. URL `http://www.eclipse.org/gmt/amw/usecases/mergeSVG/`. Last accessed August 08, 2008.

Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. doi: 10.1109/TSE.2004.23.

CPP. The C preprocessor. URL `http://gcc.gnu.org/onlinedocs/cpp/`. Last accessed August 08, 2008.

Krzysztof Czarnecki. Overview of Generative Software Development. In *Proc. Int'l Workshop on Unconventional Programming Paradigms 2004 (UPP'04)*, volume 3566 of *LNCS*, pages 326–341, Le Mont Saint Michel, France, September 2005. Springer.

Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development*, October 2008a. Special Issue on Aspects and MDE (to appear).

Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th Int'l Conf. on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008b. ACM. doi: 10.1145/1370175.1370199.

Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3): 165–184, 2008. doi: 10.1049/iet-sen:20070060.

K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, Jyväskylä, Finland, June 1997. Springer.

Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A. C. Polack. *Epsilon*. Department of Computer Science, University of York, 2008. URL: http://epsilonlabs.wiki.sourceforge.net/Book.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

Sun Microsystems. Enterprise JavaBeans Specification, Version 2.0. Final Release, August 2001.

Object Management Group. CORBA components. OMG Document, January 2008. URL `http://www.omg.org/docs/formal/08-01-08.pdf`.

Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005. ISBN 978-3-540-24372-4.

Reuseware Project Team. Reuseware composition framework webpage. URL `http://reuseware.org/`. Last accessed August 08, 2008.