Feature Modeling

Andreas Speck[1] and Elke Pulvermüller[2]

[1] Intershop Research Jena
D-07740 Jena, Germany
a.speck@intershop.com
[2] Institut für Programmstrukturen und Datenorganisation,
Universitaet Karlsruhe,
D-76128 Karlsruhe, Germany
pulvermueller@acm.org

Abstract. Component-based approaches for the software development are well-known. Most of these approaches (like CORBA and COM++) focus on the realization of the connection between the components and their interactions.
In this paper we concentrate on the missing items in the component-based software engineering: the problem how to model component composition and to validate them. We apply the component model based on the interface description with *InPorts* and *OutPorts* which allow a rather detailed definition of the components interaction. Moreover we take the term feature to name the core requirements to a component.
Features are used to drive the description of the component composition which is regarded as an combination of features expressed by logical operators. Moreover the *InPorts* and *OutPorts* describe the dynamic component interactions. The combination of *OutPorts* and *InPorts* according to the component composition rules allows to reason about the component system's dynamic behavior.

1 Introduction

The software development from components is increasingly gaining importance. Like with objects there exist different definitions for components The most well-known is given in [8]. Components and their composition are supported by various technologies such as CORBA, COM/DCOM or Java Beans. Components themself are already beyond being issue of research only. Different commercial vendors provide components as building blocks (e.g. Enterprise Beans) for industrial systems.

Now the question rises how to build up (or model, design and verify) interactions between the composed components. We consider that a component provides one feature or a set of features. The design of a system of components is driven by the desired features and their interactions [3], [5].

1.1 Component Models

There exist various component models. Most interesting are models that focus on interfaces of the components. In [4] *InPorts* and *OutPorts* are used to specify these interfaces. The protocols of the *InPort* and *OutPort* interactions are specified by automates. Such a component is depicted in figure 1.

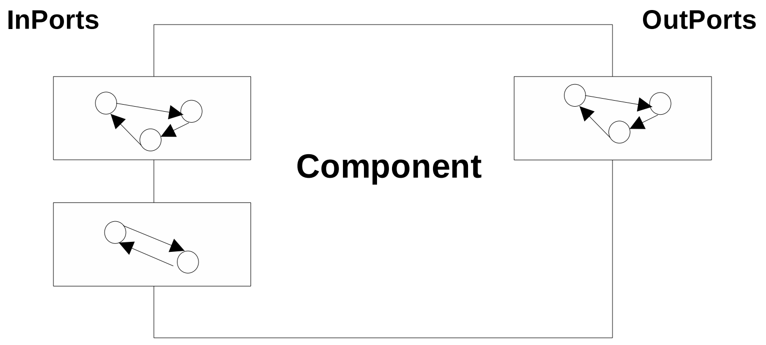

Fig. 1. Component Model with *InPorts* and *OutPorts*

The combination of components is described by the combination of the specific *InPort* and *OutPort* automates. These automates may be the base for an automatic verification of the interfaces between the components. This approach is rather old. The concept on in- and out-interfaces and the verification of their equivalence was already an issue in research area of modules [1].

1.2 Features

Features are properties of components. A component may provide a single feature or a set of multiple features. A feature itself may consist of other features.

Features are either basic features (usually representing a single method or attribute) or combined features (c.f. section 2).

The features or respectively the names of the features have to be annotated to the objects, methods or attributes of the components. In the most simple case this has to be done manually by the developer of the component as documentation of the feature. Composed features are expressed by their sub-features.

2 Feature driven Composition

Features are properties of components. In the same way as components may be combined in new larger components features may be combined too. Features may be combined form other (sub-)features. E.g. the feature *text editor* consists of several features:

– *editing algorithms* like *insert*, *undo*, *save* and *load*
– *text attributes* e.g. text which itself may contain other attributes (or features) like *words*, *sentences* or *paragraphs*

The feature *text editor* may be expressed as:

$$\begin{aligned} \textit{text editor} &= \textit{editing algorithms} \wedge \textit{text attributes} \\ \textit{editing algorithms} &= \textit{insert} \wedge \textit{undo} \wedge \textit{save} \wedge \textit{load} \\ \textit{text attributes} &= \textit{words} \wedge \textit{sentences} \wedge \textit{paragraphs} \end{aligned}$$

Of course all basic logical operations are available in order to combine features. Figure 2 shows the notation which may be applied. In the example the *SuperFeature* contains the *SubFeature A*, *SubFeature B* and *SubFeature C*:

$$SuperFeature = SubFeature\ A \wedge SubFeature\ B \wedge SubFeature\ C$$

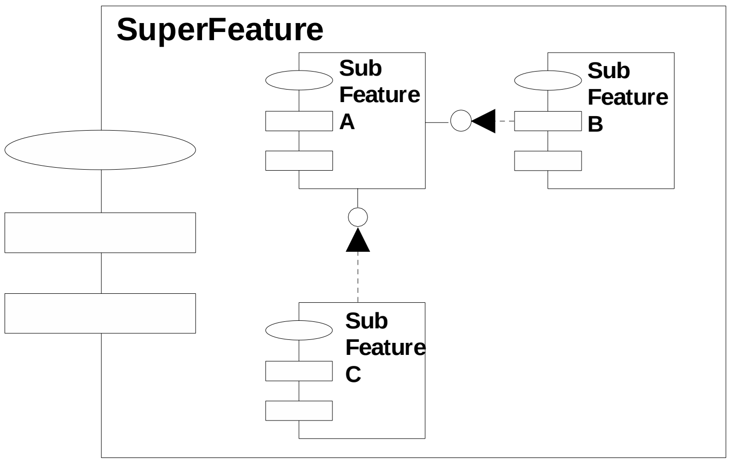

Fig. 2. Feature-driven Combination

3 Feature Validation

The application of a logic description of feature combinations enables to control and verify systems. A system may be designed as logical dependencies of different features. Such a logic design can automatically be processed and may serve as base for automatic combination tools (generators [7]).

In general there are two potential ways of verifying or validating features of a system:

1. Static Composition
   The static design may be verified by adding rules, e.g. *Feature A* requires *Feature B*, or *Feature A* excludes *Feature C*. When a specific design has to be reused the appropriate rules may simply be reused. In order to improve reuse the rules may be stored in a database.
   A more detailed description of this type of verification may be found in [3].

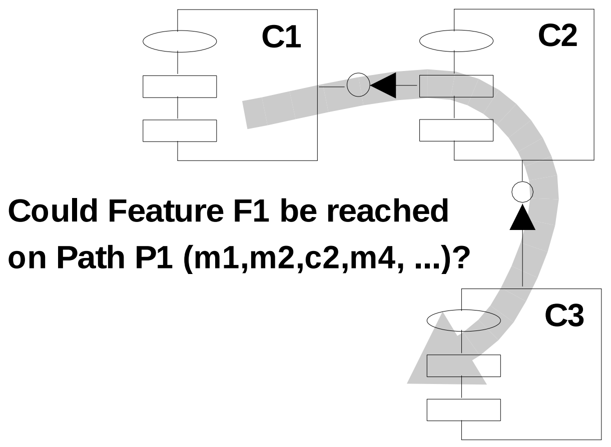

Fig. 3. Reasoning about Features Combination

2. Dynamic Behavior
   In addition to the static composition it may be interesting to reason about the dynamic interaction of a system. Figure 3 presents an example: Could a specific feature be reached when a defined sequence of input messages is given to the system?

   Other questions may be:
   – In which order are specific features passed when a message is sent to the system?
   – When there are specific alternatives in a feature system, which feature is passed when a specific condition is given by a specific message?
   – Does the system run into a dead-lock?
   – Do redundant paths exist in a system?

The validation of the dynamic behavior may be supported by tools. An approach may be the combination of the *InPort* and *OutPort* automates by additional automates representing the internal activities of a component. Component automates defined in this way may be combined in order to specify the systems behavior. A detailed description of this approach may be found in [6]. Moreover in this approach missing methods of components may be detected. Based on this an automatic adaption of the component system is possible by adding adaptors providing the missing methods.

Since it is hard to determine state charts describing the internal behavior of a component automatically we decided to parse out UML sequence charts [9]. Moreover the sequence charts may be reduced to the sequences of direct impact from the *InPort* states to the *OutPort* states.

In our approach we applied a model checking tool [2] to validate the system's dynamic interactions (c.f. figure 4). Therefore we define an *Actor* which sends

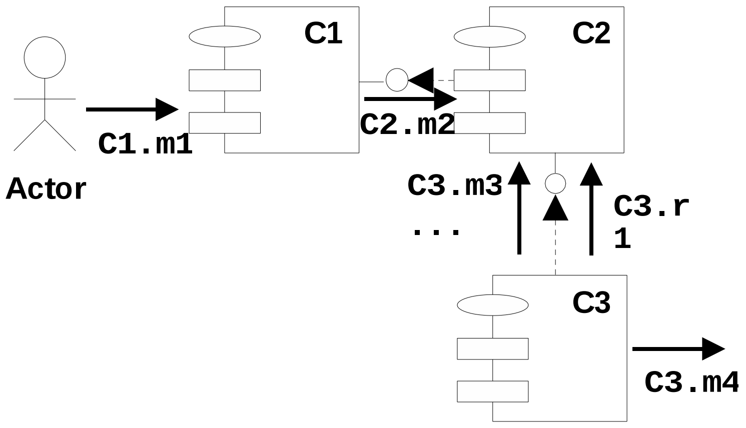

Fig. 4. Actor triggering Components Interactions

messages to the system to be validated. Within the system we determine the dependencies and communication relationships between the components. Now the model checking tool allows to reason whether a specific message or feature is triggered by message given by the *Actor*.

4 Conclusion and Future Work

In this paper we present a method to describe the design of a component system by the logical combination of features. We introduced the term feature in order to define the basic intentions of components which may be realized by methods, data or objects. The combination of this features drive the components composition.

Besides this statical view of a component system we consider the dynamic behavior as well. We consider the components' interfaces (*InPorts* and *OutPorts*) as a key issue for the specification of the dynamic interaction between components. *InPorts* and *OutPorts* are defined by automates representing their temporal actions. Within a component both may be connected by additional automates or sequence charts in order to define the components' behavior. The combination of *OutPorts* and *InPorts* of dependent components allows to reason about their interactions. In this way entire component systems may be validated automatically.

An interesting issue of our future work will be the feature composition guided by artificial intelligence. This is an improvement of the feature design data base which may help to partially reuse existing design or to detect critical combinations which do not necessarily lead to errors.

The application of AI may also help to extend the component combination approach considering not only the boolean operations conjunction, disjunction and negation but also fuzzy values in-between the range of true and false.

References

1. P. Gouthier and S. Pont. *Designing Systems Programs.* Prentice Hall, Englewood Clifs, 1970.
2. McMillan K. *Symbolic Model Checking.* Carnegie Mellon University, 1992.
3. H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering GCSE'2000*, Erfurt, Germany, October 2000.
4. A. Lauder and S. Kent. EventPorts: Preventing Legacy Componentware. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC 99)*, 1999.
5. E. Pulvermüller, A. Speck, and J. Coplien. A Version Model for Aspect Dependency Management. In *Proceedings of the Third International Symposium on Generative and Component-Based Software Engineering GCSE'2001*, Erfurt, Germany, September 2001.
6. R. Reussner. Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. In *Proceedings of the GI joint Workshop of GI AK 2.1.9 and AK 2.1.4*, Bad Honnef, Germany, May 2001.
7. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Lecture Notes in Computer Science LNCS 1445*, pages 550 – 570, 1998.
8. C. Szyperski. *Component Software.* Addison-Wesley, ACM-Press, New York, 1997.
9. W. Vanderperren and B. Wydaeghe. Towards a new Component Composition Process. Brussel, Belgium, 2001. Vrije Universiteit.