Model Superimposition in Software Product Lines

Sven Apel[1], Florian Janda[1], Salvador Trujillo[2], and Christian Kästner[3]

[1] Department of Informatics and Mathematics, University of Passau, Germany,
{apel, janda04}@uni-passau.de
[2] IKERLAN Research Centre, Spain,
STrujillo@ikerlan.es
[3] School of Computer Science, University of Magdeburg, Germany,
ckaestne@ovgu.de

Abstract. In software product line engineering, feature composition generates software tailored to specific requirements from a common set of artifacts. Superimposition is a technique to merge code pieces belonging to different features. The advent of model-driven development raises the question of how to support the variability of software product lines in modeling techniques. We propose to use superimposition as a model composition technique in order to support variability. We analyze the feasibility of superimposition for model composition, offer corresponding tool support, and discuss our experiences with three case studies (including an industrial case study).

1 Introduction

Modeling is essential to deal with the complexity of software systems during their development and maintenance. Models allow engineers to precisely capture relevant aspects of a system from a given perspective and at an appropriate level of abstraction. Initially, modeling in software development aimed at the description of single software systems. Typically, for each software system there is a set of models that describe its static structure, dynamic behavior, interaction with the user, and so on. With 'model' we refer henceforth to a concrete software artifact written in a modeling language that describes a certain facet of a software system.

Recently, researchers and practitioner have realized the necessity for modeling variability of software systems [1, 2, 3, 4]. Especially, software product line engineering poses major challenges on contemporary modeling techniques [5]. A *software product line* is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [6, 1, 7]. A *feature* is an end-user visible behavior of a software systems, and features are used to distinguish different software systems, a.k.a. *variants*, of a software product line [1]. For example, in telecommunication software, automatic callback and an answering machine are two features that are not necessarily present in all possible telecommunication systems.

There are two facets of modeling in software product lines. First, there are approaches for describing the variability of a product line, i.e., they specify which feature combinations produce valid variants [1]. Second, all variants of a product line may have models that describe them. Since the variants have usually significant overlaps in their

functionality, their models have significant overlaps, too, and it is desirable to factor out these overlaps. So, modeling languages have to take this into account.

We aim at the latter facet of modeling. The basic idea is to decompose a (structural, behavioral, etc.) model according to the features of the product line. This allows us to generate models for individual variants by composing model fragments, instead of maintaining individual models for every variant. A *model fragment* is a part of a model that covers only a feature of a system. Apart from that, a model fragment has to be syntactically complete according to its metamodel. Two things are needed for that: (1) a means to express model fragments and (2) a mechanism to compose model fragments in different combinations yielding different variants. For example, in our telecommunication system, we have a base system and two features. Each of them contains a class diagram (state diagram, activity diagram, and so on) that captures precisely the part of the complete model that is 'added' by the feature. When generating a specific telecommunication system variant, the engineer selects the desired features and then the corresponding models are composed by a generator.

The decomposition of models into fragments (features) solves two problems in modeling, *complexity* and *variability* [2, 8, 3, 9]: (1) engineers tame complexity by modeling only parts of a possibly large system, and (2) engineers do not provide models for each distinct variants of a software system but they provide model fragments for the system's features, from which the models of the variants are generated. Solutions to both problems are essential in order to increase the productivity of modeling and to let the vision of model-driven development come true.

Many approaches to feature composition rely on the technique of *superimposition* [10, 11, 12, 13, 14]. Superimposition is a relatively simple composition technique that, nevertheless, has been used successfully for the composition of code, written in a wide variety of languages, e.g., Java, C#, C++, C, Haskell, Scheme, JavaCC, and Bali [10, 14, 11]. So, naturally, the question arises whether superimposition is expressive and powerful enough for the composition of models, especially, in the face of the diverse kinds of models used today.

For the purpose of a compact discussion, we concentrate on three kinds of models supported in the unified modeling language (UML), namely, class diagrams, state diagrams, and sequence diagrams. They differ significantly in their syntax and semantics and thus cover a sufficiently broad spectrum of model elements for our discussion. On the basis of an analysis of the feasibility and expressiveness of model composition with superimposition, we have extended the feature composition tool FEATUREHOUSE for UML model composition. We used FEATUREHOUSE in three case studies on model composition: two academic product lines and a product line of our industrial partner, from which we will report our experiences. The studies demonstrate that, even though superimposition is a simple, syntax-driven composition technique, it is expressive enough for the scenarios we looked at. Furthermore, our analysis and studies reveal some interesting issues w.r.t. the trade-off between expressiveness and simplicity, which we will discuss.

In summary, we make the following contributions: (1) an analysis of superimposition as composition technique for UML models, (2) a tool for UML model composition on the basis of superimposition, and (3) three case studies (including an industrial study) on model superimposition and a discussion of our experiences.

```
1 package phone;
2 class Phone {
3   String number; String owner; int state;
4   void outCall(String num) { ... }
5 }
6 class OutCall {
7   double time; double date;
8   void makeCall(String num) { ... }
9 }
```

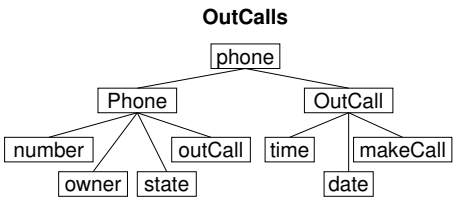

Fig. 1. Java code and FST of the feature OUTCALLS.

2 Software Product Lines and Feature Composition

A *software product line* is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [6, 7, 1]. A *feature* is an end-user visible behavior of a software system. Features are used to distinguish different *variants* of a software product line. *Feature composition* is the process of assembling and composing software artifacts (e.g., models) belonging to different features based on a user's feature selection [10, 14].

One popular approach to feature composition is superimposition. *Superimposition* is the process of composing software artifacts by merging their corresponding substructures on the basis of nominal and structural similarity. Superimposition has been applied successfully to the composition of software artifacts in different application scenarios and languages [10, 15, 16, 14].

In recent work, it has been shown that superimposition as feature composition technique is applicable to a wide variety of software artifacts, mostly source code artifacts written in different languages, e.g., Java, C#, C++, C, Haskell, Scheme, JavaCC, and Bali [10, 14, 11]. Before we explore the feasibility and expressiveness of superimposition for model composition, we explain how superimposition is commonly used for composing source code artifacts.

On the left side in Figure 1, we show a simple Java class implementing a rudimentary phone system that allows clients to make outgoing calls. On the right side, we show the abstract structure of the class in the form of a *feature structure tree* (*FST*) [13, 14]. An FST represents the essential modular structure of a software artifact and abstracts from language-specific details. In our example, the FST contains nodes representing the classes Phone and OutCall as well as their members.[1]

Now suppose we want to add a feature that allows users to receive incoming calls. In Figure 2, we show the Java code and a corresponding FST that implement incoming calls. In order to create a phone system that contains both features, outgoing and incoming calls, we superimpose their FSTs. In Figure 3, we show the result of the superimposition of the FSTs of OUTCALLS and INCALLS. The two FSTs are superimposed by merging their nodes, matched by theirs names, types, and relative positions, starting from the root and descending recursively. Their superimposition, denoted by '•', results in a merged

[1] Note that typing information is attached to each node and the actual content of the methods is hidden in the leaves of the FST.

```
1  package phone;
2  class Phone {
3    int voiceMsg;
4    void inCall() { ... };
5  }
6  class InCall {
7    double time; double date;
8    void acceptCall() { ... }
9  }
```

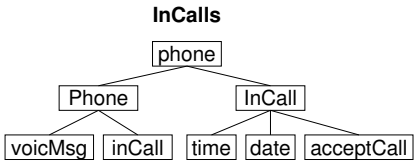

Fig. 2. Java code and FST of the feature INCALLS.

```
1  package phone;
2  class Phone {
3    String number; String owner; int state;
4    void outCall(String num) { ... }
5    int voiceMsg;
6    void inCall() { ... };
7  }
8  class OutCall {
9    double time; double date;
10   void makeCall(String num) { ... }
11 }
12 class InCall {
13   double time; double date;
14   void acceptCall() { ... }
15 }
```

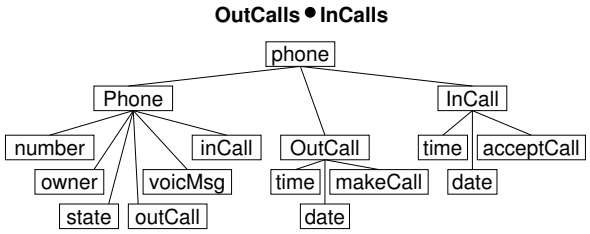

Fig. 3. Java code and FST of the superimposition of OUTCALLS and INCALLS.

package phone containing all three classes Phone, OutCall, and InCall. The class Phone is interesting as it contains the union of the members of its instances in the features OUTCALLS and INCALLS.

Assuming further features like CALLBACK or ANSWERING, we have an SPL from which several telecommunication system variants can be generated, e.g., INCALL • ANSWERING, OUTCALL • CALLBACK, or OUTCALL • CALLBACK • INCALL • ANSWERING. Since not all combinations form meaningful variants, additional constraints define which combinations are valid [17, 18].

While superimposition has been used successfully in software product lines [10, 19, 14] and component systems [16], it is not a general technique for all kinds of composition scenarios. A limitation is that the structures of two software artifacts being composed have to obey certain similarities that guide the composition. Accidental name clashes cannot be detected and possibly necessary renamings cannot be performed automatically since the semantics of the underlying artifacts is not considered. We discuss the implications of this limitation for our case studies in Section 6.

As said, it has been shown that superimposition is applicable to a wide variety of software artifacts written in different languages [10, 14]. However, superimposition imposes several constraints on the target language: (1) the substructure of a feature must be a hierarchy of modules, (2) every module (submodule, ...) of a feature must have a name, and (3) the name of a module must be unique in the scope of its enclosing module (that is, a module must not have two submodules with identical names). Typ-

diagram	element	name	variants (excerpt)	composition
class	package	identifier	—	superimposition
	entity	instance, class	class, interface, type	superimposition
	attribute	type, variable	valued attributes	replacement
	operation	signature	—	replacement
	relationship	optional identifier	generalization, association	replacement
state	state	identifier	—	superimposition
	transition	optional identifier	—	replacement
	start/stop state	—	—	replacement
sequence	role	instance, class	—	superimposition
	lifeline	—	—	concatenation
	interaction	—	—	—

Table 1. An overview of UML model elements and their composition.

ically, these constraints are satisfied by most programming languages, but also other (non-code) languages like XML or grammars align well with them [10, 20, 14]. In the next section, we analyze whether and how superimposition is capable of being used for model composition, especially for the composition of UML models.

3 Analysis of Superimposition as Model Composition Technique

In this section, we analyze whether UML class diagrams, state diagrams, and sequence diagrams, can be decomposed into features and composed again to create complete models corresponding to the variants of an SPL. We do not address layouting issues but only structural composition. In Table 1, we give an overview of our findings, which are conveyed in the remaining section. The table shows for each diagram type (Column 1) the elements that can be composed (Column 2), how elements are identified – if at all (Column 3), possible variants of an element (Column 4), and the composition method, i.e., how two elements are composed to form a new element (Column 5). Note that the difference between superimposition and replacement is that with superimposition the corresponding substructures of two elements are superimposed recursively. With replacement one element substitutes the other completely. Concatenation is used only in sequence diagrams, which is explained below.

Class Diagrams. An UML class diagram consists of a set of packages, entities, and relationships between these entities. Entities are displayed by boxes and may denote plain classes, implementation classes, association classes, interfaces, types, and so on. A package may contain several further packages and entities. Each package and entity has a unique name (possibly consisting of an instance and a class name) in the scope of its enclosing package as well as a type that corresponds to its syntactical category or stereotype (e.g., «class» or «interface»). Relationships are displayed by different kinds of arrows and denote inheritance, aggregation, composition, and so on. Relationships do not necessarily have unique names but may have annotations such as cardinalities or message names.

We illustrate the superimposition of class diagrams by the example of our phone system. In Figure 4, we show from right to left the class diagrams of OUTCALL, INCALL, and their superimposition OUTCALL • INCALL.

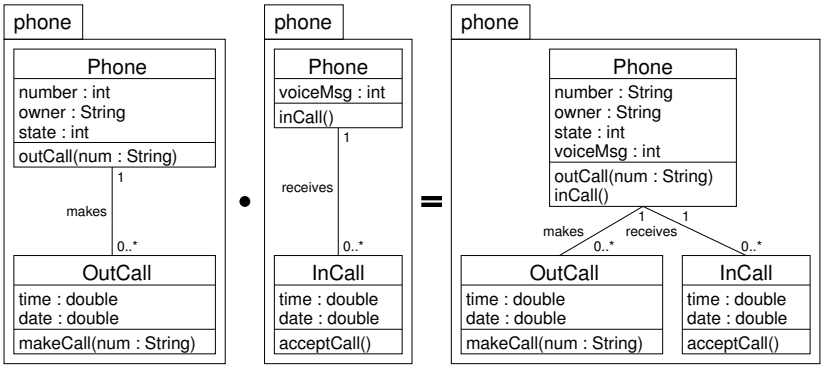

Fig. 4. Superimposition of the class diagrams of OUTCALLS and INCALLS.

During superimposition, entities are matched by name and type. That is, the class Phone in package phone is merged with the other class Phone in package phone stemming from another feature. Entities of different types or from different packages are not composed. When composing two entities, the union is taken from their members, much like in our Java example in Figure 3. If there are two members with the same name (and type), one replaces the other, whose order is inferred from the feature composition order.

Relationships cannot be merged in every case by name since they do not always have names. Implicitly, we assume that each unnamed relationship has a unique name (possibly automatically assigned by a tool) – thereby satisfying the second and third constraint of Section 2. Hence, in our example, both of the two associations from the features OUTCALLS and INCALLS appear in the superimposed diagram. If two relationships have identical names, one replaces the other, much like with members. Other types of entities and relations are treated similarly to the ones explained above.

In summary, the superimposition of class diagrams is straightforward as it resembles the superimposition of FSTs of object-oriented code. Of course, more complex, possibly semantics-driven, composition rules are possible, e.g., for the composition of cardinalities, but for our use cases, a simple replacement of the involved relationships is simpler and sufficient, as we will discuss in Sections 5 and 6.

State Diagrams. An UML state diagram consists basically of a set of states and a set of state transitions. States have unique names and may have inner states. So, they can be composed by name. As with packages, inner states must have unique names in the scope of their enclosing state in order to be superimposed recursively. A state transition may have annotations and a unique name. In case there is no unique name, we assign one, as with relationships in class diagrams. Other elements of state diagrams are handled simi-

larly to states and transitions; if they have names (e.g., signals), they are treated like states; if they do not have unique names (e.g., choices), they are treated like unnamed transitions.

The start and stop states of a state diagram deserve a special attention. Each state diagram must have exactly one start and possibly one stop state. For composition that means that the start and stop states of one feature replace the start and stop states of the other feature.

In Figure 5, we illustrate the composition of two state diagrams that describe the behavior of our phone system. The state diagrams of OUTCALLS (left) and INCALLS (middle) are superimposed (shown on the right). Essentially, the feature INCALLS adds a new transition from the state StandBy to the state Calling. In our case studies, we found more complex examples in which also compound states and multiple start and stop states are involved.

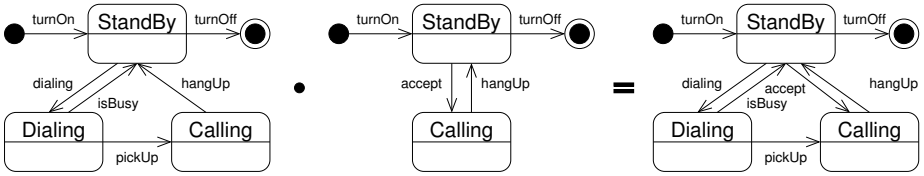

Fig. 5. Superimposition of the state diagrams of OUTCALLS and INCALLS.

In summary, the superimposition of state diagrams is similar to class diagrams, except the treatment of start and stop states. States correspond roughly to classes and transitions correspond roughly to relationships. All other elements such as signals and choices can be assigned to one of these two categories. As with class diagrams, more complex composition rules are possible, e.g., for composing choices, but we found our rules are sufficient, as we will discuss in Sections 5 and 6.

Sequence Diagrams. An UML sequence diagram consists of a set of roles that represent time lines of objects and different kinds of interactions that represent the timely interplay between objects. Roles have unique names that consist of object and/or class name. Hence, they can be composed by name, much like classes. Furthermore, each role has a lifeline – displayed from top to bottom – that is the source and destination of interactions with other roles, which are denoted with arrows. Along the lifelines, the time proceeds from top to bottom. Interactions have usually annotations (e.g., operation names and guards) but do not have unique names. Although there may be names, these do not need necessarily to be unique. For example, along its lifeline, a role may interact with another role via multiple interactions that have identical names (e.g., by invoking an operation multiple times), but that occur at different points in time. Hence, we assign unique names (our tool does), much like we do with unnamed relationships in class diagrams.

The composition of two roles with the same name is done by adding the interactions of the second role to the ones of the first role. In Figure 6, we show the composition of the sequence diagrams of the features OUTCALLS (left) and INCALLS (middle). Phone

is the only role that is defined in both features. In the superimposed lifeline (right), the Phone's lifeline of feature INCALLS is added below the corresponding lifeline of feature OUTCALLS, which is de facto a concatenation.

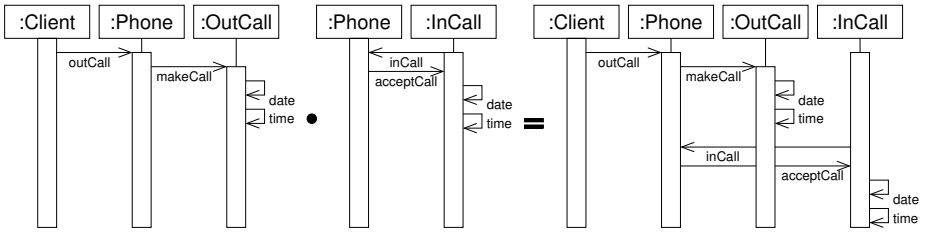

Fig. 6. Superimposition of the sequence diagrams of OUTCALLS and INCALLS.

While this approach works for our example, this kind of composition of lifelines may be too restricted. What would we have to do if we want to add the interactions of INCALLS before or somewhere in the middle of the ones of OUTCALLS? While the former case would be possible by reversing the composition order of OUTCALLS and INCALLS, the second case cannot be implemented using superimposition. This problem has been observed in the context of other programming languages and several solutions have been proposed, e.g., splitting features or injecting hooks [14] (see also Sec. 6).

In summary, sequence diagrams can be composed like class diagrams, where roles correspond to classes and interactions correspond to relationships, except that the order of interactions matters in the composition of lifelines. As with class and state diagrams, more complex composition rules would be possible, e.g., for composing guards, but we found our rules are sufficient (see Sec. 5 and 6).

4 Tool Support

Based on our analysis of the feasibility of superimposition for the composition of UML models, we have extended our tool FEATUREHOUSE[2] to enable it to compose UML diagrams. FEATUREHOUSE is a software composition tool chain that relies on feature structure trees and superimposition [14]. In Figure 7, we show the architecture of FEATUREHOUSE. The central tool is FSTCOMPOSER which language-independently superimposes FSTs stored in a proprietary data structure. At certain points during superimposition special composition rules are applied, such as replacement and concatenation, which we have implemented as specified in Table 1.

Furthermore, FSTGENERATOR generates for each language (1) a parser that translates artifacts written in that language to FSTs and (2) a pretty printer that creates an artifact out of an (composed) FST. FSTGENERATOR receives as input an annotated FEATUREBNF grammar (an extended BNF format with annotations that control superimposition) of the language whose artifacts are going to be composed. The details of

[2] `http://www.fosd.de/fh`

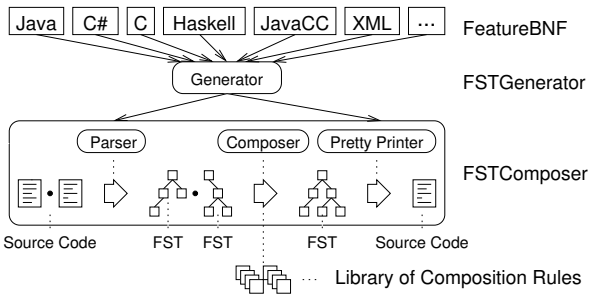

Fig. 7. Architecture of FEATUREHOUSE.

this generation step are out of scope of this paper and explained elsewhere [14]. We have implemented the rules of superimposing UML diagrams as explained in Section 3.

Technically, we use the XML metadata interchange 1.2 format (XMI) [21] as an representation of UML diagrams and ArgoUML[3] for creating, displaying, and editing UML diagrams stored in XMI files. Points in favor of XMI are that XMI files are machine-readable by FEATUREHOUSE and can be transformed easily into FSTs. In a first step, we have developed a parser manually (without a generation step) that translates class, state, and sequence diagrams stored in XMI files to FSTs and a pretty printer that translates the (composed) FSTs back to XMI files. In a second step, we have extended FSTGENERATOR by the ability to generate a parser and pretty printer automatically on the basis of an annotated grammar, which was in our case an XML schema document. Interestingly, the annotated grammar plays the role of the metamodel of the models we consider. This raises the question if other more common metamodels such as Ecore could be used instead, which we address in further work.

A typical development cycle consists of four steps: (1) ArgoUML is used to design model fragments belonging to different features; (2) the visual model representations are exported to XMI using ArgoUML's standard export facilities; (3) FEATUREHOUSE is used to compose multiple model fragments based on a user's feature selection; (4) ArgoUML is used to view the composed result. We have used FEATUREHOUSE (and ArgoUML) in three case studies, which we will explain next. For simplicity, we assume that the feature selections passed to FEATUREHOUSE are valid. Existing tools like GUIDSL [18] can be easily used with FEATUREHOUSE to ensure the validity of a feature selection.

5 Case Studies

We have explored the feasibility and practicality of FEATUREHOUSE for model composition by means of three case studies. Specifically, we wanted to know whether such a simple approach like superimposition is expressive enough to compose models in practice. All models (XMI and PNG files) of all case studies can be downloaded from

[3] `http://argouml.tigris.org/`

the Web.[4] In order to protect the intellectual property of our industrial partner, we have made the models of the gas boiler system (third case study) anonymous to some extent.

class diagrams:

	class		**member**			**relationship**		
	all	**extended**	**all**	**added**	**replaced**	**all**	**added**	**replaced**
ACS	12	4	30	7	0	8	3	1
CMS	7	3	28	9	6	4	2	0
GBS	22	8	24	31	61	8	12	4

state diagrams:

	state					**transition**		
	all	**extended**	**compound**	**start**	**stop**	**all**	**added**	**replaced**
ACS	11	2	2	3	1	25	8	0
CMS	16	7	2	3	3	26	18	2

sequence diagrams:

	role/lifeline		**interaction**		
	all	**extended**	**all**	**added**	**replaced**
ACS	8	5	3	14	0
CMS	5	6	11	9	0

all: number of all elements in the product line; **extended**: number of all elements that have been extended, possibly multiple times; **added**: number of all elements that have been added to a given program; **replaced**: number of all elements that have been replaced by other elements; **compound/start/stop**: number of compound/start/stop states;

Table 2. An overview of UML model elements and their composition.

Audio Control System. As a first, simple case study, we have designed an audio control system (ACS). ACS is a small product line consisting only of three features: a basic amplifier, a remote control, and a compact disk player. We designed the structure and behavior of ACS with class, state, and sequence diagrams. Overall, we have developed three class diagrams, two state diagrams, and three sequence diagrams, i.e., not for every feature there is a distinct state diagram. From the ACS product line four different variants can be generated which results in different class, state, and sequence diagrams that have been composed by the model fragments of the selected features. In Table 2, we show some numbers about the models of this case study.

The decomposition of ACS into features and their subsequent composition in different combinations was straightforward. This may be due to the simplicity of the case study and/or that we designed it with features in mind. The dominant activity of features was to add new elements and to extend existing elements with subelements, such as to extend classes with members or lifelines with interactions. This case study provides a good canonical example for the feasibility of model composition by superimposition.

[4] `http://www.fosd.de/mc/ICMT2009.zip`

Conference Management System. As a second case study, we have decomposed the conference management system (CMS) of [22] into features. The initial version of CMS contained only class diagrams, decomposed into four features: user management, submission system, submission verification procedure, and review process. Based on the informal description of the authors of CMS [22], we complemented the class diagrams with state and sequence diagrams. Although more are possible, we have composed four concrete variants of CMS. Overall, the decomposed version of CMS consists of four class diagrams, four state diagrams, and four sequence diagrams. Like ACS, the decomposition into features and the composition of features in order to generate variants was mostly straightforward. However, CMS is more complex than ACS and was designed by a third party and not designed as an SPL – so it is a more unbiased example. An interesting property of CMS is the use and superimposition of compound states, as can be seen in Table 2. These are superb use cases for superimposition.

Gas Boiler System. As a third case study, we have composed different variants of a gas boiler system (GBS). GBS was developed by a customer of our industrial partner, the IKERLAN Technology Research Centre,[5] Mondragon, Spain, for serving as embedded control software of gas boilers. The developers at IKERLAN have refactored the initial versions of GBS into a software product line. Furthermore, they have used modeling techniques, in particular, UML class diagrams, to describe the structure of the overall system. In contrast to our approach, they have developed a complete class diagram in which model elements of all features are merged, and they use annotations to maintain the relationship between model elements and features.

Based on their annotations, we have decomposed the complete class diagram into 29 class diagram fragments that correspond to the 29 features of GBS. Despite the complexity of this case study, the (de)composition was relatively simple because information about features and variants was available from our partner. Mostly, features add new classes and extend existing classes by new members and relationships. One interesting issue was how to model the variability inside a method of a class. The problem is that class diagrams do not expose details about method bodies and the developers did use notes or other kinds of annotations for that. In their initial approach, they simply annotated a method to be associated to two or more features. This is not possible with superimposition. Instead, we have modeled this kind of *intra-method variability* by introducing a corresponding class and member to every feature that affects this member. For example, if feature A and B both affect the implementation of a method m contained in class C, then the class diagrams of both features define a class C with method m.

5.1 Summary of Experiences

We summarize our experiences in the following list:

- Mostly, the decomposition of models into features and the composition of features for the generation of model variants was straightforward. The predominant "activities" of features were to add new elements such as classes, states, or roles, and to

[5] http://www.ikerlan.es

extend existing elements, e.g., by adding new attributes, operations, relationships, transitions, or interactions. Usually, the addition of new elements and the extension of existing elements occurred together in order to connect the newly introduced elements in some way to the existing elements. Extensions at a finer grain, such as changing or extending a guard or an annotation, did not occur, even not in the third-party case studies.
- The fact that two models need to obey certain similarities in their structure in order to be superimposed caused no problems in our case studies. However, for modeling product lines from scratch, as in the case of the ACS study, it is necessary to plan carefully the structure of the base models and their subsequent extensions applied by features. A decomposition of existing models, as in the GBS case study, naturally leads to models that can be superimposed again in different combinations.
- The problem that roles in sequence diagrams can only be extended by adding new interactions at the end or in front of their lifelines did not occur in our case studies. Usually, the features that extend lifelines add sequences of interactions that are semantically distinct from the existing interactions. Of course, in other cases this problem may be more daunting and has to be explored then.
- We found some convincing use cases of nested model elements, e.g., compound states in CMS, that take advantage of the recursive superimposition process.
- Intra-method variability, i.e., the situation in which multiple features affect the implementation of a single method, occurs in several situations in our industrial case study. The reason is that the developer already thought about the implementation of the model. We handle this case by introducing a corresponding model element into each feature that is involved.
- Cases for the replacement of existing elements did not occur frequently. We found only use cases for relationships in which two identical relationships have been composed. The reason was readability, i.e., both class diagram fragments were easier to understand with the relationship in question.

6 Discussion: Simplicity vs. Expressiveness

Superimposition is a comparatively simple composition technique. Not least this is rooted in its aim at generality. Based on our experience with superimposition of source code artifacts, we have explored if it is applicable to model composition. Since models are mostly of a hierarchical structure, superimposition is a good match, as it is indicated by our examples and case studies. Due to our focus on software product lines and feature-based (de)composition, the input models naturally have similar structures so that they could be easily superimposed. In other scenarios, such as multi-team development, additional refactorings may be necessary.

However, applicability is not the only criterion. Is superimposition expressive enough to compose models in practice? Many model transformation and composition techniques aim at more powerful, fine-grained, and semantics-based composition models [23, 24, 25, 26, 3, 2, 27]. With these techniques, even elements such as guards or cardinalities can be composed, elements can be renamed and changed in manifold ways, and semantic constraints check the correctness of a transformation/composition. Superimposition

is syntax-driven without semantic checks and there are not many ways to change an element (replacement and concatenation). However, erroneous compositions have been rejected by analyzing a feature model that defines the valid feature combinations of an SPL [18]. Additionally, experts from our industrial partner checked the results of our compositions for correctness. But it is certainly desirable to automate this process using constraint-based techniques, e.g., [25]. However, at least for our case studies, superimposition was expressive enough to satisfy the composition demands of the considered applications scenarios in SPL development. Especially, the industrial setting of the GBS study indicates that, with superimposition, we can go a long way.

Important to note is that superimposition is no technique for the integration of arbitrary models. It is useful for decomposing models into features, where each feature-related model fragment has a certain structural similarity with other model fragments in order to be superimposed. Automated renamings, rebindings, and refactorings are not supported. Thus, superimposition is not the "silver bullet", but useful at least in the context of feature composition and software product lines. Also the success of superimposition for the composition of code is a motivation for our work. We believe that a general model for feature composition (for source code, models, documentation, makefiles, test cases, etc.) helps the programmer to tame complexity and to gain insight into the software since all software artifacts are treated uniformly during composition.

Our work suggests that there is a trade-off between expressiveness (fine-grained, semantics-based composition) and simplicity (superimposition). Of course, we cannot judge for one or the other, nor infer an ideal mixture of both. This trade-off has been discussed for years in the programming languages community and led to several interesting approaches (just think of the difference in complexity of Scheme and Haskell); we believe that our work can help to initiate a discussion about this issue for modeling languages and model composition mechanisms.

7 Related Work

Our approach to superimposition on the basis of FSTs has been influenced from AHEAD [10]. AHEAD is a model for feature composition that emphasizes language independence of superimposition. Work on AHEAD has claimed that superimposition should be in principle applicable to a wide variety of software artifacts. Work on FEATUREHOUSE has further developed the concept of language-independent superimposition on the basis of FSTs [14]. Composition of UML models is one application scenario that has been analyzed in this paper.

Aspect-oriented modeling (AOM) aims at separating model elements that belong to different concerns. Although features and concerns are not entirely identical concepts [19], decomposing models into features and compose them again on demand is very similar to the aims and procedures of AOM. In AOM, usually, different model fragments are "woven" using certain more or less explicit composition rules. As there is a multitude of different AOM approaches, we use three representative examples to explain the difference to our approach. For example, in the Theme/UML approach, models can be composed by user-defined composition rules that also include to some extent the superimposition by name and type [8]; in the aspectual scenario approach

different scenarios are merged and state machines are generated [2]; Jezequel has shown how aspects can be woven into sequence diagrams [26], which is more flexible but also more complex than superimposition. In contrast to these sometimes very different AOM approaches, superimposition on the basis of the FST model is very simple, general, and language-independent, as it can be used to compose so different artifacts like Java, Haskell, and UML models in a uniform way.

Heidenreich et al. propose an approach to enhance modeling languages with composition capabilities based on the meta model [28]. They allow two models to be composed at predefined hooks, which is essentially an interface-based approach. Superimposition is simpler as it merges models by nominal and structural similarities without interfaces, which was sufficient for our case studies. Superimposition is inherently non-invasive whereas the interface-based approach requires a proper preparation.

Several model merge tools, e.g., the Epsilon Merging Language [27] or the ATLAS Model Weaver [3], support diverse kinds of transformations of models. These tools could be used to implement model composition by superimposition. However, superimposition is simpler but also less expressive than other merge tools. As with programming languages, there is a trade-off between simplicity and expressiveness which we find largely unexplored in model transformation.

Boronat et al. present an automated approach for generic model merging from a practical standpoint, providing support for conflict resolution and traceability between software artifacts by using the QVT Relations language [22]. They focus on the definition of an operator *Merge* and apply it to class diagrams integration; other model types are not considered. We have shown that our approach is also applicable to different kinds of models.

FeatureMapper [9] and fmp2rsm [29] are tools with which developers can annotate models with features. Annotations help programmers to overview and understand how individual features influence the structure and behavior of a software system. Annotation is conceptually related to the decomposition of models into features. Annotation and decomposition deserve further investigation since they have complementary strengths and weaknesses [30]. In a recent study, FeatureMapper and FEATUREHOUSE have been used to annotate/decompose entity-relationship-diagrams [31]. However, the focus of this work was on tailoring database schemas and not on model composition.

Feature-oriented model-driven development is an approach that ties feature composition to model-driven development [5]. The core of this approach is a theory based on category theory that relates transformations that stem from feature composition to transformations that stem from model refinement. In their case study, they use the Xak tool [20] to compose state machines written in a domain-specific language, which is related to FEATUREHOUSE but not language-independent.

The package merge mechanism of UML is related to superimposition [32]. It combines the content of two packages. Package merge is used extensively in the UML 2 specification to modularize the definition of the UML 2 metamodel and to define the four compliance levels of UML 2. However, package merge is not applicable to states and roles, and defines many specific composition rules. Our implementation is much simpler and extends to other kinds of models.

Finally, there is a relationship to domain and variability modeling techniques [1,17,4]. While these techniques are used to model the variability of software systems in terms of

features and their relationships, we support the derivation of different model variants based on feature composition.

8 Conclusion

We have explored the feasibility and expressiveness of superimposition as a model composition technique. Our analysis and case studies indicate that, even though superimposition is syntax-driven and quite simple, it is indeed expressive enough for the systems we looked at – even in the context of a real-world, industrial case study. We offer a tool that is able to compose UML class, state, and sequence diagrams in the form of XMI documents via superimposition. In further work, we will extend the analysis, the tool, and the case studies in order to support further kinds of UML and non-UML models. Furthermore, we will explore the connection of our metamodel (annotated grammar) to other metamodels for superimposition. Finally, we will experiment with specific (semantics-based) composition rules for individual model elements in order to explore the trade-off between simplicity and expressiveness and to combine them with superimposition eventually.

Acknowledgments

Apel's work was funded partly by the German Research Foundation, project #AP 206/2-1. Trujillo's work was co-supported by the Spanish Ministry of Science & Education under contract TIN2008-06507-C02-02.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, CMU (1990)
2. Whittle, J., Araujo, J.: Scenario Modelling with Aspects. IEE Software **151** (2004) 157–172
3. Jossic, A., et al.: Model Integration with Model Weaving: A Case Study in System Architecture. In: Proc. Int. Conf. Systems Engineering and Modeling, IEEE CS (2007) 79–84
4. Sinnema, M., Deelstra, S.: Classifying Variability Modeling Techniques. Inf. Softw. Technol. **49** (2007) 717–739
5. Trujillo, S., Batory, D., Díaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: Proc. Int. Conf. Software Engineering, IEEE CS (2007) 44–53
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
7. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer-Verlag (2005)
8. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley (2005)
9. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Controlled Visualisations in Software Product Line Engineering. In: Proc. Int. Workshop Visualisation in Software Product Line Eng., Lero, University of Limerick (2008) 335–342
10. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30** (2004) 355–371

11. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proc. Int. Conf. Generative Programming and Component Engineering, Springer-Verlag (2005) 125–140
12. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. Int. Symp. Foundations of Software Eng., ACM Press (2004) 127–136
13. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Proc. Int. Symp. Software Composition, Springer-Verlag (2008) 20–35
14. Apel, S., Kästner, C., Lengauer, C.: FeatureHouse: Language-Independent, Automatic Software Composition. In: Proc. Int. Conf. Software Engineering, IEEE CS (2009)
15. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int. Conf. Software Engineering, IEEE CS (1999) 107–119
16. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology **41** (1999) 257–273
17. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
18. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Proc. Int. Software Product Line Conf., Springer-Verlag (2005) 7–20
19. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering **34** (2008) 162–180
20. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Proc. of Int. Conf. on Web Engineering, Springer-Verlag (2007) 473–478
21. Grose, T., Doney, G., Brodsky, S.: Mastering XMI. OMG Press (2002)
22. Boronat, A., Carsí, J., Ramos, I., Letelier, P.: Formal Model Merging Applied to Class Diagram Integration. Electron. Notes Theor. Comput. Sci. **166** (2007) 5–26
23. Klein, J., Helouet, L., Jezequel, J.: Semantic-Based Weaving of Scenarios. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2006) 27–38
24. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: An Algebraic View on the Semantics of model Composition. In: Proc. Europ. Conf. Model Driven Architecture – Foundations and Applications, Springer-Verlag (2007) 99–113
25. Czarnecki, K., Pietroszek, K.: Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In: Proc. Int. Conf. Generative Programming and Component Engineering, ACM Press (2006) 211–220
26. Jezequel, J.M.: Model Driven Design and Aspect Weaving. Software and Systems Modeling **7** (2008) 209–218
27. Kolovos, D., Paige, R., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: Proc. Int. Conf. Model Driven Engineering Languages and Systems, Springer-Verlag (2006) 215–229
28. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. Trans. Aspect-Oriented Software Development (2009)
29. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Proc. Int. Conf. Generative Programming and Component Engineering, Springer-Verlag (2005) 422–437
30. Kästner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, Dept. of Informatics and Mathematics, University of Passau (2008) 35–40
31. Siegmund, N., et al.: Bridging the Gap between Variability in Client Application and Database Schema. In: Proc. Conf. Datenbanksysteme für Business, Technologie und Web, Gesellschaft für Informatik (2009) 297–306
32. Group, O.M.: Unified Modeling Language: Superstructure (Version 2.1.1) (2007)