An Approach to Addressing Entity Model Variability within Software Product Lines

Joerg Bartholdt
Siemens AG
Corporate Technology
Architecture, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
joerg.bartholdt@siemens.com

Roy Oberhauser
Aalen University
Computer Science Dept.
Beethovenstr. 1,
73430 Aalen, Germany
roy.oberhauser@htw-aalen.de

Andreas Rytina
Siemens AG
Corporate Technology
Architecture, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
andreas@rytina.net

Abstract

Software Product Line (SPL) engineering is one approach for addressing customization and variability for products. However, current approaches and research, while often addressing feature modeling and component variability, insufficiently address difficulties and additional complexity with respect to entity model variability, which negatively impacts various software qualities, such as correctness, reusability, maintainability, testability, and evolvability.

This paper presents a solution approach with an integrated mechanism providing a consistent view to capture data variability in entity models, while hiding and decoupling components from superfluous data elements via adapter generation. An eHealth SPL case study is presented supporting adapter generation with differential entity conversion. The results show that with this approach, entity model variability can be effectively addressed and desirable software qualities preserved.

1. Introduction

SPL Engineering (SPLE) seeks to foster a systematic reuse of software components for different but similar software products (usually, products in the same domain). The general approach is to capture the commonalities and variability of the products in the product line and split the development into domain (commonalities) and application (additional individual features for the final product). Products are built by integrating the common artifacts (usually a platform) and optionally configuring them with product-specific artifacts [11] [14].

Significant work and various methodologies are well known for domain analysis and variability modeling for SPLs with a focus on features, e.g., Feature-Oriented Domain Analysis (FODA) methodology [9], FeatuRSEB [8], PuLSE [2], and others. Typical feature models in SPLs allow for many ($\sim 10^x$) possible permutations. Considering that an artifact may influence the entity model (e.g., adds new entities or relations), all artifacts must be able to handle multiple entity variants, although they themselves make no use of the available differences. Yet the aforementioned methodologies do not sufficiently support and address variability in the entity models. The Orthogonal Variability Model's (OVM) [14] does go beyond features to addressing variability in artifacts, but is an abstract approach missing a notation that can be used by automation for entity models.

The Approach for Entity Model Variability (AEMV) described in this paper shows an integrated mechanism for SPLE to consistently view and edit the entities within the domain model, capture the variability, as well as shield artifact developers from extraneous differences.

To motivate and demonstrate the features of AEMV, a case study in the medical domain for an eHealth SPL specified by a third party served as the research basis - and is presented in simplified for this paper. In the following section the scenario and solution requirements are presented. In section 3, the AEMV solution is applied, and then evaluated in the subsequent section, especially with regard to variation scalability. Related work is discussed in section 5, followed by the conclusion.

2. Scenario and Requirements

There is an increasing market demand in e-Health for integrated medical information systems and solutions, with globalization in the market and customization demands even spanning national boundaries. The difficulties for developing and supporting such systems become apparent in time-to-market, labor costs, and error-proneness when aligning and supporting the various entity models needed for such systems. To support a variety of markets, a product line approach allows the medical information platform customer to select arbitrary features as add-ons to the base product, e.g., date-definition, document repository, information service, or internationalization. This entails various challenges, among them that the overall product instance-specific entity model changes with the features selected.

For example, a medical care information system might provide an Information Service that allows doctors to automatically print informational medical service offers or recommendations for patients based on their medical history. The address of the patient would be required by the component and retrieved, (e.g., via `patient.getAddress()`). For international customization to the United Kingdom (UK), a resident may have 2 addresses for tax purposes, thus perhaps requiring the component to make a different call (e.g., `patient.getAddress().getFirst()`).

This problem might be addressed by another component that returns the required patient information, however the exchange of patient objects between components would be impeded.

2.1. Requirements

The deficiencies in the examples above illustrate the following requirements that are imposed on the solution to cope with variability in entity models:

1) Modeling of the entity objects in the solution space must be consistent and provided in a central view (analogous to the feature tree in the problem space that shows a central view of the variants of the product line). The individual products must be derived from this model.

2) Developers of artifacts shall be shielded from the effects of the many possible variants on their code (API and structure of the domain objects) while retaining the compile-time safety that getter/setter navigation in the domain object model guarantees. This includes the demand for loose-coupling not only for the functionality of components, but also for the data exchanged between those services.

3) Interoperability of artifacts shall be supported automatically over the SPL lifetime even if the development takes place at different times and disparate locations, thus implying support of multiple versions of the artifacts.

Although this case study comes from the e-Health domain, the issues are representative and applicable to entity variability in SPLs in general.

3. Solution

The AEMV Process is a unique UML standards-based approach for entity modeling usable with common MDSD tooling, integrated with feature modeling, and supporting desirable software qualities during SPL development. AEMV starts in the Domain Engineering phase with requirements analysis (see Fig. 3.1). This is used to create a Feature Model (e.g., using FMP [1]). An Entity Model is created in the Unified Modeling Language 2 XMI (XML Metadata Interchange) that includes variations. The Application Engineering phase uses the configuration in FMP as well as the Entity Model as inputs (e.g., using openArchitectureWare (oAW) [12]) to create an Entity Model Instance, from which the required code artifacts are generated (e.g., using Xpand).

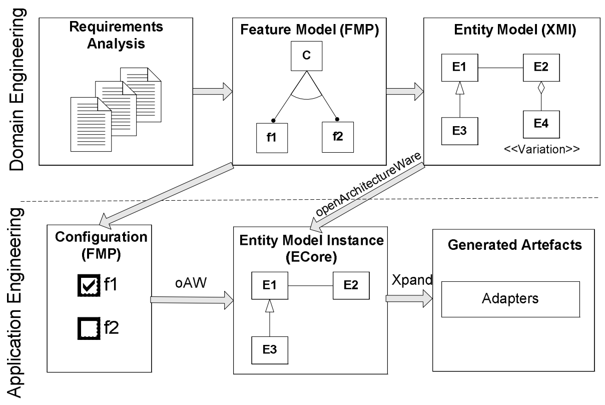


Figure 3.1. AEMV Process

AEMV addresses three types of variability: *positive* - adding new fields, entities, relations; *negative* - eliminate fields, entities, relations; and *structural* - variations in type, cardinality or naming of elements.

Unified Modeling Language 2.x (UML2) class diagrams were selected for modeling due to the extensibility via stereotypes (in contrast, e.g., to the entity-relationship diagram) and the plethora of tools available to process the UML model further.

Fig. 3.2 shows the (reduced) Feature Model (FM) for this domain using the Czarnecki-Eisenecker notation [4]. The entity model is described in UML class diagram as shown in Fig 3.3.

The variation points in the solution space are marked with the stereotype `<<Variation>>` on class, association or member level. An appended condition describes which features in the feature model must be selected in order for this part to appear in the domain model of the product instance. Note that Boolean expressions are allowed, e.g., `Feature1 AND NOT Feature2`.

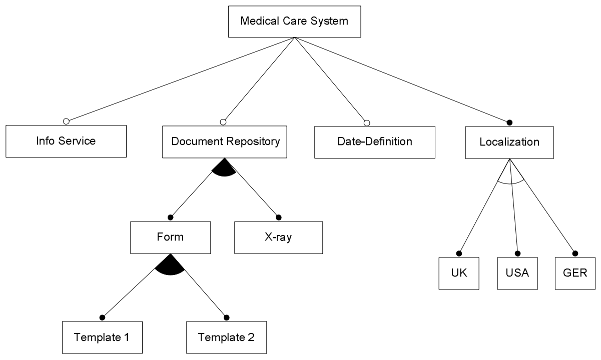


Figure 3.2. Feature Model

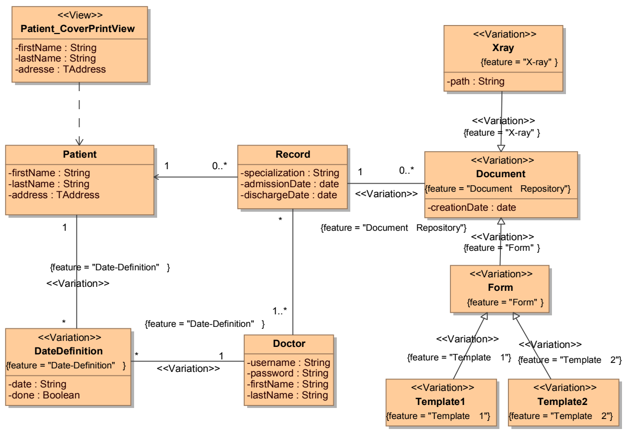


Figure 3.3. Entity Model

Selecting the X-ray and the Date-Definition feature results in the following product instance entity model:

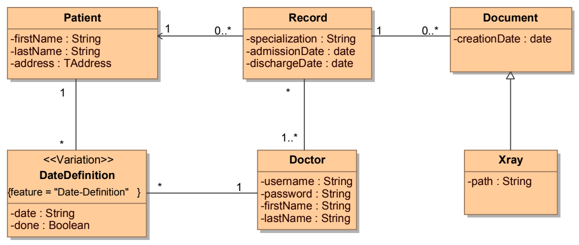


Figure 3.4 Entity Instance Model

The artifact generation is done by oAW's template engine Xpand. The oAW workflow, which invokes the Xpand-template, produces the Java interfaces and the API-Implementation. The resulting API is presented below in simplified form:

Listing 1

```
public interface Patient
{
   String getFirstName();
   void setFirstName(String value);
   String getLastName();
   void setLastName(String value);
   List<Record> getRecords();
   void addRecords(List<Record> values);
   void removeRecords(List<Record> values);
   ...
}

public interface Records
{
    String getSpecialization();
    void setSpecialization(String value);
    Date getAdmissionDate();
    Date setAdmissionDate(String value);
    ...
}
```

Deselecting all features results in the common entity model. This example depicts positive variability as only elements where added.

When customizing the entity model to support patients who reside in the UK, structural variability is introduced as the address changes its cardinality to 2. This would invalidate all code that uses the (simple) Patient type since the API changed, thus introducing the need for an adapter to shield those components from the differences. Manually written adapters place an additional burden on the developer: besides the initial development, they must be kept consistent with the changes in the entity model over time. AEMV models those adapters together with the entity model and generates the code normally automatically. This is a step towards a consistent view on the entity model over the whole product line over time and it allows the exchange of entities between components with different views on those artifacts.

The following diagram depicts a subset of the entity model:

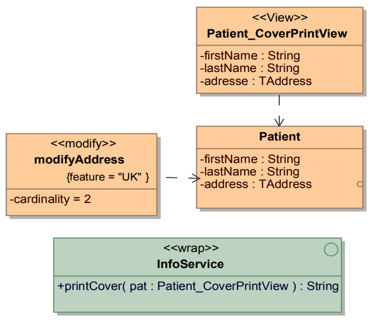


Figure 3.5. Entity Model Subset showing Adapter and View

Note that the arrow from the `modifyAddress` class points to the member `address`, not the class, so that it is clear which member will be modified `<<modify>>`.

Selecting the UK feature in the feature model results in the following entity model instance:

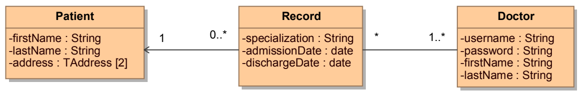


Figure 3.6. Entity Model Instance

Adapters are based on the original entity object of the product instance and provide a more stable view on the entity for components that only require a subset. The adapters are generated automatically at least for members with the same name. For more difficult conversions, only the getter and setter are generated - the implementation must be done manually.

The adapters provide multiple data views to components and utilize a common entity model, thus conversions are required at runtime. In Fig. 3.7a the conversion relationships between views to support the differing projected views desired by components is shown, with a maximum number of unidirectional converters required being n(n -1) where n is the number of views required. Fig 3.7b shows that the maximum number of converters required can be reduced to a linear 2(n-1) if the direct conversions between projected views are avoided and only conversions to and from a complete view are utilized (a star topology). This incurs a higher runtime cost of two conversions (once to the common view and then to the desired view) vs. only one, but benefits maintenance and evolution due to the reduced number of adapters. The runtime impact is dependent on the number of elements and complexity of conversion.

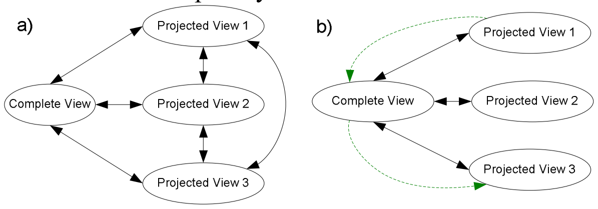


Figure 3.7. Projected View Conversions

The resulting code of the adapter is shown below in Listing 2. The manually added conversion code to convert from any source view to the target Patient_CoverPrintView is shown in bold, and includes the conversion via the Complete View.

Listing 2

```
public class InfoServiceAdapter
 implements InfoService
{
  private InfoService adaptee;

  public InfoServiceAdapter
   (InfoService adaptee) {
       this.adaptee = adaptee;
  }
  public void  printCover
       (Patient_CoverPrintView view) {

    Patient_CoverPrintView adaptedView =
        new  Patient_CoverPrintView();
    this.convert(view, adaptedView);

    this.adaptee
        .printCover(adaptedView);
  }
  private void convert(User srcView,
           Patient_CoverPrintView targetView)
  {
    targetView
      .setFirstName(srcView.getFirstName());
    targetView
      .setLastName(srcView.getLastName());
    targetView.setAddress(
       srcView.getAddress().getFirst()
     );
  }
}
```

4. Evaluation

Due to the use of code generation techniques, the impact of the variations and the use of the adapters at runtime is relatively inconsequential. View conversion of data where necessary, e.g., from one format to another, is currently a manual programming task and thus the runtime impact dependent on the conversion complexity. However, due to the large set of possible permutations and the reliance on MDD, variation scalability measurements were made to determine the impacts of the variations for development time usage of AEMV.

The measurements were performed on a AMD Athlon XP 2400+ (2GHz) PC with 3GB RAM running Microsoft Windows XP Pro SP2, Java JDK 1.6, Eclipse 3.3, openArchitectureWare 4.2, and the Eclipse Modeling Framework 2.3. All measurements were performed 3 times and the averages presented.

For the first set of measurements, the transformation time using oAW from an XMI Entity Model file containing variations to an Entity Model Instance XMI (all variation points applied based on features) was measured as shown in Table 4.1 and Fig. 4.1. A nearly linear correlation between a change in the number of variation points and the generation time was measured as the number of features was held constant, and an increase in the number of features also showed a nearly linear increase in the generation time. This result is explained by the iterations in the generator code implementation for each variation point and for each feature. Varying the number of Boolean conjunctions up to 20 for a variation point made no significant difference due to other inherent overheads.

Table 4.1. Entity Model Instance Transformation Time (ms) for Features and Variation Points

Number of Variation Points	Total Number of Features		
	300	600	900
50	2771	5281	9141
100	4429	9696	17781
150	6416	14219	26078

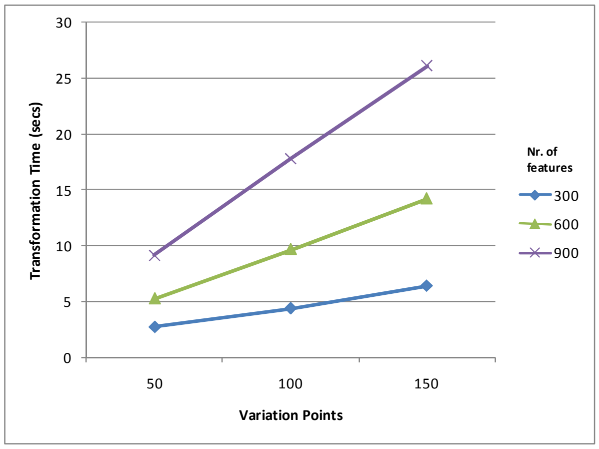


Figure 4.1. Entity Model Instance Transformation Time vs. Variation Points and Features

A second set of measurements concerned the generation of adapters. Each of the different variability types was tested and, as expected, no noticeable difference in generation time occurred based on the negative, positive, or structural variability types. In AEMV, each adapter for an entity can support multiple projected views. The Lines of Code (LOC) generated in support of the conversion between views increased in the same percentage to the number of views, as expected due to the 2n relation resulting from the complete view basis for all conversions. The maintainability of the conversions is thereby supported. The generation time required for adapters with multiple views is shown in Fig. 4.2., showing a nearly linear increase as the number of adapters or views increase. The generation time for this scale appears reasonable for development usage in current SPLs.

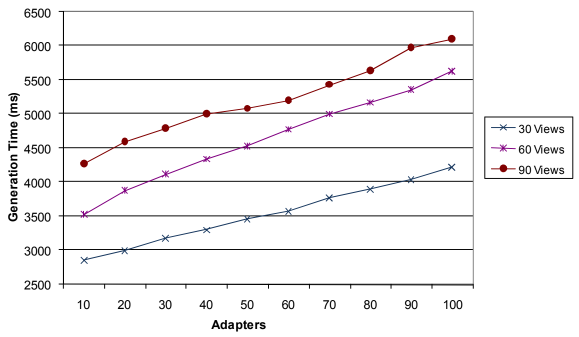


Figure 4.2. Generation Time vs. Number of Adapters

In summary, the development-time variation scalability and performance of AEMV with current tooling for industrial use is shown to be practicable.

5. Alternatives and Related Work

UML2 introduced the "package merge" feature [13] and its usage for SPLs has been evaluated [17][10]. Class extensions (e.g., additional members) can be modeled in a separate package that must have a merge association with the base package. The mapping is done on class name equality. Package merge is not suitable for the requirements described in section 2, because it scatters the variation point over multiple packages. Thus the number of packages explodes and does not scale well with the number of features. Package merge cannot model negative or structural variability that is needed for requirement 2.

Explicit dependencies. Each extension to the entity model could be presented by a separate entity component and explicitly used by a functional component. The entity components retrieve the necessary elements to form their view on a domain entity. Communicating with other components reduces the obligation for the receiving component to retrieve their view of the entity again.

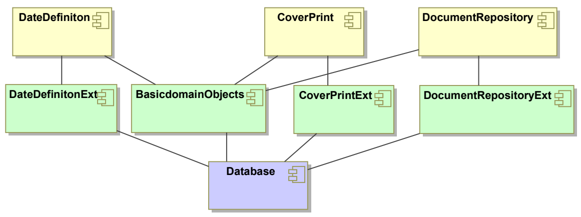
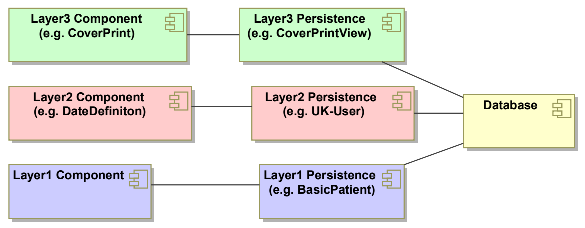


Figure 5.1 Entity Model with Explicit Dependencies

Drawbacks include the numerous calls for database retrievals per component due to a lack of sharing, as well as interpretation difficulties when components transmit or receive entities via references or value objects. If transactions are considered or the services are remote, this solution is infeasible.

Layering. Similar to the Decorator design pattern [6], components are grouped in layers that correspond to the level of enrichment of the entity. Per layer, one definition of each entity exists. If a component of a higher layer needs an entity, it asks the persistence component of that layer to retrieve it. The persistence component routes this request down the layers and extends ("enriches") the data, converting the entity into its own layer entity model.



Layer3 Component (e.g. CoverPrint)
Layer3 Persistence (e.g. CoverPrintView)
Layer2 Component (e.g. DateDefiniton)
Layer2 Persistence (e.g. UK-User)
Database
Layer1 Component
Layer1 Persistence (e.g. BasicPatient)

Figure 5.2 Entity Model with Layered Extensions

Multiple calls have to be executed to provide the layer 3 component with the entity and there is insufficient support for transaction handling.

Both of the above alternatives have the disadvantage that the entity model is not presented consistently and that knowledge is spread over at least some layers or even individual components. This assumes that a strict layering is even possible.

5.1. Related Work

Other approaches for SPL variability in domain models include the conceptual framework SPLIT [3], where additional UML stereotypes, e.g., <<variabilityMechanism>> and <<variationPoint>>, are used for specifying variable elements. SPLIT does not, however, integrate an abstract feature view as does AEMV, and each variation point and the corresponding variants requires a separate class which can cause issues in lucidity for large SPLs.

In [7] the PLUS (Product Line UML-Based Software Engineering) extends UML to model variability and commonality using stereotypes and primarily subclassing. While entities are mentioned, the wrappers described are intended for database access and do not support all variation types and multiple view and entity versions for components as addressed in AEMV. The extension of PLUS with AEMV would provide a more comprehensive solution for SPL UML techniques.

In MDD-AO-PLE [15][16] and similar related aspect-oriented SPLE work, the application of techniques to SPLs are investigated for addressing cross-cutting variability. While this work has not specifically addressed the difficulties described in this paper for entity models, the combination of these techniques with AEMV could be synergistic, e.g., to address common data view format conversions in adapters.

Work with regard to SPL component evolution support includes [5], where a multi-team decentralized SPL variability modeling approach is described, supporting the merging of model fragments. However, it does not address versioning of entities and component usage and lacks UML support.

6. Conclusion

Given the inadequate integration and specific support for MDSD entity modeling and variability in current SPL approaches and research, AEMV contributes a clear process and UML standards-based approach for entity modeling that can be utilized by common MDSD tooling, is integrated with feature modeling, and supports desirable software qualities during SPL development. Its scalability with regard to features and variation points is linear and likely to be sufficient for typical current SPL development.

Based on a concrete e-Health case study, the issues addressed in entity model variability are nevertheless applicable to SPLs in general. By choosing UML-based stereotypes to model variability, portability and utilization of multiple MDSD implementations for modeling, model-to-model transformations, and generation can be used. Via the generation of projected views of the entity for a component in its current version, correctness is supported in that extraneous entity attributes are hidden. Maintenance and evolvability are supported by shielding component developers from changes via projected views and the complete view superset. Reusability of components is supported since no tight coupling directly to other components via entities occurs. Furthermore, interoperability is supported via the projected views. Component testing is simplified since component testing can be done without knowledge of other existing components in the individual product instances.

Future work includes the addition of a conversion language for somewhat complex conversions in adapters (e.g., concatenation and regex-split). Support of dynamic runtime variation including adaptation and binding of component views with database migration is another area to be investigated. Additionally, the integration of aspect-oriented approaches, optimization for object tree transfers, and greater automatic adapter entity conversion code generation are promising.

7. References

[1] Antkiewicz, M. and Czarnecki, K. 2004. FeaturePlugin: feature modeling plug-in for Eclipse. In Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange (Vancouver, British Columbia, Canada, October 24 - 24, 2004). eclipse '04. ACM, 67-72.

[2] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J. “PuLSE: a methodology to develop software product lines.” In *Proceedings of the 1999 Symposium on Software Reusability* (SSR '99). ACM, 122-131.

[3] M. Coriat et al, “The SPLIT method”, In *Proceedings of the First International Software Product-Line Conference* (SPLC-1) 2000.

[4] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addison–Wesley, May 2000. ISBN 0201309777.

[5] D. Dhungana, Thomas Neumayer, Paul Gruenbacher, Rick Rabiser, "Supporting the Evolution of Product Line Architectures with Variability Model Fragments," *wicsa*, pp. 327-330, Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), 2008

[6] E. Gamma, R. Helm, R. Johnson , J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995, ISBN 0-201-63361-2.

[7] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures,* Addison-Wesley, 2005, ISBN 0201775956.

[8] Griss, M. L., Favaro, J., and Alessandro, M. d. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th international Conference on Software Reuse* (1998). ICSR. IEEE Computer Society.

[9] K. Kang, S. Cohen, et al “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” Software Engineering Institute, Carnegie Mellon University, 1990.

[10] Laguna, M. A., González-Baixauli, B., and Marqués, J. M. “Seamless development of software product lines”. In *Proceedings of the 6th international Conference on Generative Programming and Component Engineering* (GPCE 2007). ACM, New York, NY, 85-94.

[11] F.J. v.d. Linden, K. Schmid, E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, 2007, ISBN 3540714367.

[12] M. Voelter, A. Haase, B. Kolb, S. Efftinge, “Introduction to openArchitectureWare 4.1.2”. In *Proceedings of the Model-Driven Development Tool Implementers Forum* (MDD-TIF07 Workshop at TOOLS EUROPE 2007).

[13] OMG, “UML 2.1.2 Superstructure Specification”, OMG doc# formal/07-11-02, 2007.

[14] K. Pohl, G. Böckle, F.J. v.d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, 2005, ISBN 3540243720.

[15] M. Voelter and I. Groher, “Handling Variability in Model Transformations and Generators”, in *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling* (DSM’07), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.), Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2.

[16] M. Voelter, and I. Groher, “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development”, in *Proceedings of the 11th International Software Product Line Conference (SPLC)*, 2007.

[17] A. Zito and J. Dingel. "Modeling UML 2 Package Merge With Alloy". *Proc. of the 1st Alloy Workshop* (Alloy '06). Portland, Oregon, USA. November, 2006.