# Bridging the Gap between Variability in Client Application and Database Schema

Norbert Siegmund[1], Christian Kästner[1], Marko Rosenmüller[1],
Florian Heidenreich[2], Sven Apel[3], and Gunter Saake[1]

[1]School of Computer Science, University of Magdeburg, Germany
[2]Department of Computer Science, Dresden University of Technology, Germany
[3]Department of Informatics and Mathematics, University of Passau, Germany
[1]{nsiegmun,ckaestne,rosenmue,saake}@ovgu.de,
[2]florian.heidenreich@tu-dresden.de, [3]apel@uni-passau.de

**Abstract:** Database schemas are used to describe the logical design of a database. Diverse groups of users have different perspectives on the schema which leads to different local schemas. Research has focused on view integration to generate a global, consistent schema out of different local schemas or views. However, this approach seems to be too constrained when the generated global view should be variable and only a certain subset is needed. Variable schemas are needed in software product lines in which products are tailored to the needs of stakeholders. We claim that traditional modeling techniques are not sufficient for expressing a variable database schema. We show that software product line methodologies, when applied to the database schemas, overcome existing limitations and allow the generation of tailor-made database schemas.

## 1 Introduction

Database client applications become more and more customizable and variants can be tailored for specific use-cases and user requirements. Prominent techniques to achieve variability and create custom variants are software product lines [BCK98, PGvdL05], component systems [Szy02] and plug-in based frameworks [JF88]. However, an increasing demand on variability in the client application requires new methods to tailor also the underlying database schema. In prior research, only few researchers addressed this issue, e.g., by means of view integration [SP94, Par03, BLN86] or view tailoring [BQR07]. In practice, usually a fixed global schema for all variants of the client application leads to a gap between variability in client application and database schema. The database does not match the variability of the application.

As a running example, we use a trivial software product line of a vendor of university management systems. This system can be configured for different clients, and different variants can be generated and shipped to the customer. For example, the client application can be delivered as standard variant to support course offerings and a basic book recommendation system, as extended variant which replaces book recommendations by full library support, or as an enterprise variant with connectors to existing systems for room planning.

By combining different components of the system, thousands of variants can be generated. The problem is that these variants require different underlying database schemas, which must be flexible accordingly.

A standard approach is to create a single database schema that contains all elements that might ever be needed by any variant of the client application. However, in large applications with high variability there are several problems of the mismatch between variability of the client application and the database schema. First, if variability is high, the global database schema will be large and complex. It does not scale. Second, the maintainability is reduced, because it is difficult to understand and change a global schema that is fixed for all variants. For example, tables not needed and maintained in most variants are still present in the global database schema. Furthermore, the lack of separation of concerns and, thus, the suboptimal traceability of requirements to database schema elements complicate development and evolution. This may even hinder application developers to introduce new variants in the application. Finally, variability in applications can be high, there can be millions of variants; this does not scale for one global database schema.

To address these problems and align the database schema with the variability of the client application, we propose to tailor database schemas according to user requirements. We present two software engineering techniques that are useful depending on the development and deployment scenario. (1) *Physically decomposed* database sub-schemas can be composed for a target variant. (2) A *virtually decomposed* schema (an annotated global schema) can be used to remove unneeded elements for a target variant. We offer two tools that allow developers to tailor database schemas using physical or virtual decomposition.

## 2 Software Product Line Engineering

A *software product line (SPL)* is a group of products sharing a common, managed set of features that satisfy specific needs of one domain and that are developed from a common set of core assets in a prescribed way [BCK98, PGvdL05]. For instance, the university management software described above can be considered as an SPL. Different variants in an SPL are distinguished in terms of features, where *features* are distinguishable, end-user visible characteristics of a system or domain [CE00]. For example, the variants "basic" and "extended" of our university SPL are distinguished in that variant "basic" does not have feature *Library* while variant "extended" has this feature. The overall goal of SPLs is to systematically reuse software artifacts for different software products.

Commonly, a feature model is used to describe variability of an SPL [CE00]. Feature models are represented as a tree and connections describe relationships between them. For example, a feature connected by an empty dot is optional, one connected with a filled dot is mandatory, and arcs are used to denote alternatives. Additional cross-tree constraints are possible as well. Optional and alternative features introduce variability into an SPL. In Figure 1, we show a subset of the feature model for our university SPL with already 32 possible variants. It is easy to imagine that with further features, that thousands or millions of variants are possible. This is common in SPLs today.

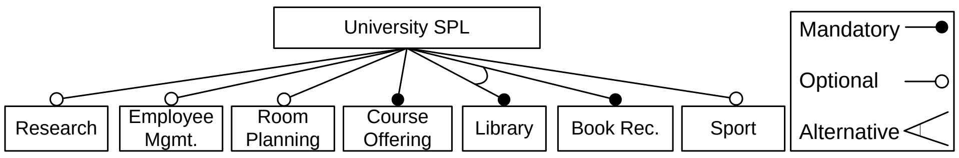

Figure 1: Simple Feature Diagram of a University CMS SPL.

Software product line engineering involves two processes: domain engineering and application engineering. While, in traditional software development, only a single application is developed, an SPL covers an entire domain. Thus, *domain engineering* starts with domain analysis, i.e., analyzing commonalities and differences between different products of a domain. As a result, a feature model describing the variability is created. Now, on a domain level, this feature model can be implemented. That means there is an implementation artifact or generator for every feature.

Creating a single variant in this domain is called *application engineering*. As a first step, the requirements of this variant are analyzed. For example, an engineer records the requirements of a specific university. Based on this requirement analysis, features of the software product line are selected. In the case that not all requirements can be covered by the SPL's features, developers must return to domain engineering and add an according feature and implementation artifact. Based on a feature selection the desired product can be assembled from the SPL's implementation artifacts. The concrete assembly process depends on the implementation mechanism (e.g., code generation [CE00], composition [BSR04, ALS08], preprocessors [KAK08, PGvdL05]).

## 3 Variable Database Schemas

Before we explain our solution, we will outline the variability problem in more detail and show which workarounds exist in current applications to overcome the problem.

### 3.1 Problem Statement

To illustrate the problem of database schema variability, we show an excerpt of our university SPL's schema as a common *entity relationship (ER)* diagram [Che76] in Figure 2. The schema models entities needed for the application's features *Course Offering*, *Room Planning*, *Library*, and *Book Recommendations* from Figure 1. Roughly the connection is as follows: in the "basic variant" with only *Course Offering* and *Book Recommendation* features, only the entities *Catalogue*, *Course*, and *SBook* are required. If feature *Room Planning* is selected, additionally entity *Room* and attribute *Capacity* of *Course* are needed, and finally entities *Book*, *Author*, and *Book Copy* are required for the *Library* feature.

Although this schema shows only a small excerpt, it is easy to imagine that large and

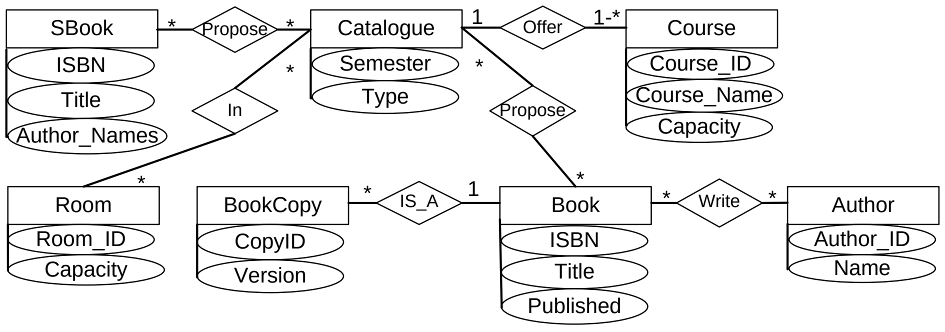

Figure 2: Database schema of the university SPL (excerpt).

complex schemas are difficult to maintain, because they merge elements required for different features. A developer who is interested only in a single feature must read and understand the entire schema including many elements not relevant for her task. Even attributes such as *Capacity* might be used only in application variants with certain features.

The problem described above for optional features gets even worse in case of alternative features. In our university example, we have two alternative variants of book recommendations for a course. In the “basic variant” *Book Recommendation*, the lecturer uses a simple web formular to enter author, title and ISBN of books she recommends. In the more advanced variant, the information are taken directly from the library (in the case the university bought the “extended variant” with library support, i.e., the *Library* feature is selected). In Figure 2, this was modeled with two alternative entities *SBook* for the “basic variant” and *Book* (and *Author* and *BookCopy*) for the extended library variant, because the same name *Book* could not be used for both, even though only one of them is needed at a time (as specified by the feature model, Fig. 1). This makes the schema hard to understand and to maintain.

## 3.2 Previous Solutions

In this section, we present an analysis of two common solutions used in practice that use mainly workarounds to solve solve the mentioned problems. The first example is taken from the web content management system Typo3 (`http://typo3.org`), which can be extended and configured with a large amount of plug-ins (e.g., for guest books, for forums, for picture galleries). Dependencies between plug-ins are specified in their description and can be considered as feature model. In this system, plug-ins may extend the database schema with additional tables or attributes. The second example is taken from a social network web-project called webuni.de (several hundreds of thousands lines of PHP, HTML and SQL code, evolved and extended over 6 years) that aims at building a series of social networks that are each local for a certain town. However, for different instances of the social network, different functionality is provided, e.g., article system, forums, calendar module, or connector to specific university management systems are selected per instance.

The software is modularized and modules can be configured flexibly for each town, however the underlying database schema for all variants is the same.

In both cases the client implementation is already modularized in terms of features, but the database schema is not. In Typo3, plug-ins can provide extensions to the database schema on SQL level (new 'create table' scripts). Because of the absence of a conceptual modeling infrastructure each extension defines their own 'create table' statements. This results in consistency problems, as the global name space is not checked (different definitions for the same table are possible) and foreign keys between tables of different plug-ins are not created. Instead of foreign keys, consistency checks are implemented as application logic.

In the social network software, all modules use the same global schema. This way, an extension for a single town requires to change all local databases and external plug-ins that require additional tables are not possible. Developers working on a single module always have the burden of understanding the entire database schema, including attributes of entities required only be few modules. This makes learning the framework difficult for new developers. Alternative features were hardly used in this project which might be caused by the involved complexity to support this. Finally, in local installations there may be many unused schema elements, which still need to be maintained by local administrators.

### 3.3 Our Solution

We aim at a proper separation of concerns already on the level of conceptual modeling. We want to be able to trace features not only to the implementation level as already done in software product line engineering, but also to database schema elements. In the target schema for a specific application variant, only the required database schema elements should be included. The basic idea is to decompose the schema in terms of features. Decomposition means the mental process of structuring a complex problem into manageable pieces. Following recent research in software engineering [KAK08, HKW08], we propose two principle approaches to decomposition, namely virtual and physical decomposition.

**Virtual Decomposition.** In a virtual decomposition, schema elements are not physically separated into different files, but only annotated. That is, schema elements belonging to features (such as entity *SBook* belongs to the *Book Recommendation* feature) are left intermixed inside a single schema, but it is highlighted which elements belong to which schema. To generate a variant for a given feature selection, the annotations are evaluated and elements that are not necessary are removed. The resulting schema in the deployed variant is a subset of the global database schema. A virtual decomposition can easily be adopted if there is already an existing database schema. In this case, the existing global schema is annotated with features to allow the tracing of the schema elements to the features, and to allow a generation of tailored variants.

A standard approach for annotations on code level is to use *#ifdef* statements of the C preprocessor. In more recent approaches [KAK08, HKW08] separately stored annotations and background colors on the representation layer are used. The advantage is that the

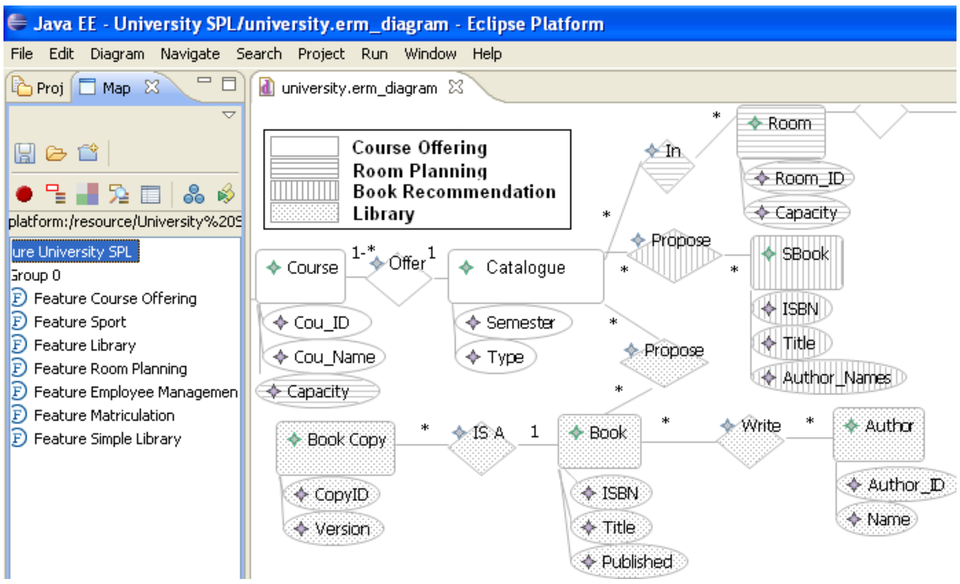

Figure 3: Eclipse plugin to realize annotations of database schemas.

developer can easily distinguish between the modeling artifacts and information regarding features. We have implemented an Eclipse-based ER modeling tool that can be used with FeatureMapper [HcW08] to annotate database schemas with features. FeatureMapper supports annotations of all models that are based on the Eclipse GEF/EMF frameworks, e.g., UML models or statechart models. We extended Eclipse with a new Ecore-based ER model editor and integrated it with FeatureMapper. Features can be shown in the user interface by different background colors, additional to the formal underlying mapping model. To achieve a separation of concerns, the editor provides views that show only schema elements of individual features, or previews of the variant model, already during development. This way, although there is no physical decomposition, it is still possible to reason about features or variants in isolation.

In Figure 3, we show a screenshot of the university SPL database schema as annotated in our editor, based on the description in Section 3.1. For the printed version of this paper, we used different background patterns to highlight features. We can even make fine-grained annotations as to annotate the *Capacity* attribute with the *Room Planning* feature.

**Physical Decomposition.** In some scenarios a virtual decomposition is not suitable. For example, when plug-ins can be provided by third parties as in the Typo3 example, an annotated global database schema is not feasible. In such case a physical decomposition is useful. Schema elements are stored in separate files, one file per feature. To generate a schema for a specific variant, those files corresponding to the feature selection are composed. The advantage of this approach is that a feature can be traced not only to the application's implementation, but also to the according sub-schema in a single file.

Feature: *Course Offering*

```
entity Catalogue { Semester, Type };
entity Course { Cource_ID, Course_Name);
relationship Offer between Catalogue (1) and
      Course (1-*);
```

Feature: *Library*

```
entity Book { ISBN, Title, Published };
entity Author { Author_ID, Name };
entity Catalogue {};
relationship Propose between Catalogue (*)
      and Book (*);
relationship Write between Author (*) and
      Book (*);
...
```

Feature: *Book Recommendation*

```
entity Book { ISBN, Title, Author_Names };
relationship Propose between Catalogue (*)
      and Book (*);
```

Feature: *Room Planning*

```
entity Room { ... };
entity Course { Capacity };
relationship ...;
```

Figure 4: Physical Decomposition.

The composition mechanism can be explained best in a textual representation[1]. In Figure 4 we show a simple textual representation of the schema in Figure 2 that has been physically decomposed into four files according to our features. For composition, the according files are selected and superimposed. Superimposition is a language-independent composition mechanism that has been applied to various code and non-code languages and was formalized in an algebra [AL08, ALMK08]. We can compose all combinations (except composing *Library* with *Book Rec.* as specified in the feature model, see Fig. 1). In our case, composition is simple because the schema language is simple: All elements are assembled in one file, and elements with the same name are merged (attributes are concatenated with removing duplicates, cardinalities are composed with a special algorithm). Entities can also be repeated if desired. The resulting schema for one variant contains the superset of all entities, relations and their attributes. For extended ER models [CRYG07, Bad04, GJ99], further extensions of the composition mechanisms are necessary.

We implemented the composition mechanism for the textual representation as an extension of the general software composition tool suite FeatureHouse [AL08]. The extension required 13 lines of code, all remaining implementation can be reused as described in [AL08]. With this tool it is possible to compose schema fragments; this makes a physical decomposition according to features feasible, and directly supports alternative features.

## 3.4 Discussion

Both approaches – virtual and physical – allow to decompose a database schema according to a feature model. They overcome the mismatch that conceptual database schemas are usually not aligned with the variability of their client applications. We can tailor a database schema exactly to the requirements of the customer in line with the customized application.

[1]Nevertheless, a graphical representation in the ER editor, or the native XMI-based storage format are possible as well. The same mechanism can also be applied to physical schemas or even SQL scripts. In fact, we implemented transformations between various different representations.

Both approaches have different strength and weaknesses [KAK08]. Therefore, we adapted both to support ER modeling. First, a virtual decomposition is well suited for decomposing existing database schemas. It makes minimal changes to the development process and can be adopted easier in an industrial scenario [CK02]. In contrast a physical separation is well suited for projects started from scratch, for projects with many alternative features, and projects where third-party developers are involved (as in the Typo3 project). Based on both approaches, automatically checking all possible variants for consistency (well-formedness) is possible [CP06], but outside the scope of this paper.

We offer tool support for both approaches and made them available online (`http://fosd.de/er/`). With this contribution, we want to raise attention to variability in conceptual modeling and stimulate further discussions and development of according modeling tools.

## 4 Related Work

Czarnecki and Pietroszek presented a methodology for a mapping features to models [CP06] which was extended and implemented as Eclipse plug-in by Heidenreich et al. [HKW08]. This idea forms the basis for our virtual decomposition approach. In contrast, physical decomposition is rooted in the idea of feature-oriented programming [Pre97, BSR04] and aspect-oriented programming [KLM$^+$97], in which software is synthesized by composing code units. We adapted these ideas to ER modelling.

Rashid proposed using aspect-oriented programming to support database schema evolution in object-oriented databases [Ras03]. While he also identifies a lack of variability in database schemas, he concentrates only on evolutionary changes and not on the derivation of different variants of the database schema.

View integration or view merging are well studied mechanisms in database research [SE06, SP94, BLN86]. The main focus lies on consistency checking, covering different, incomplete, and overlapping aspects in local views to retrieve one valid global view or schema. In contrast, we concentrate on generating a number of tailor-made schemas by refining sub-schemas to allow variability and customizeability.

Another approach for data tailoring is presented by Bolchini et al. [BQR07]. Their approach aims at generating and composing views to tailor the data of an underlying large database schema for a given context. In contrast to our approach, Bolchini et al. target context dependent data retrieval and do not tailor or change the underlying schema.

Furthermore, work on model composition is highly related to a physical decomposition. Hermann et al. defined on algebraic level an operator for model composition [HKR$^+$07]. Their theorems prove that syntactic and semantic property preserving model compositions can be achieved. In addition, Sabetzadeh et al. use also SPL concepts to describe different variants of models [SSEC07] in the automotive domain. They use views on function nets to model different features of the system. While they describe model composition on an abstract level, we use feature models to enforce the consistency of the schema composition. Moreover, we can refine already existing schemas which was not considered in their research.

## 5 Conclusions

With increasing variability in client application, there is a growing mismatch between variable applications and conceptual database schemas, that are usually fixed. We presented two solutions based on software product lines techniques to overcome this mismatch by providing a tailor-made database schema for variable applications. Virtual decomposition maps model elements of one global database schema to features of the domain, so that different variants can be generated. Physical composition creates distinct sub-models that can be composed. This reduces complexity for individual developers and allows a proper separation of concerns also for conceptual modeling.

In further work, we plan to compare both approaches in more detail in a practical case study. Furthermore, we aim at other database artifacts as physical schemas or SQL scripts, and plan to implement consistency checks that ensure well-formedness for all variants.

## References

[AL08] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Software Composition*, pages 20–35, 2008.

[ALMK08] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proc. Int'l Conf. on Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 36–50, 2008.

[ALS08] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[Bad04] A. Badia. Entity-Relationship modeling revisited. *SIGMOD Record*, 33(1):77–82, 2004.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Longman, 1998.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[BQR07] C. Bolchini, E. Quintarelli, and R. Rossato. Relational Data Tailoring through View Composition. In *26th Intl. Conf. on Conceptual Modeling (ER)*, volume 4801 of *LNCS*, pages 149–164, 2007.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Che76] P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[CK02] P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[CP06] K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 211–220. ACM, 2006.

[CRYG07] G. Chen, M. Ren, P. Yan, and X. Guo. Enriching the ER model based on discovered association rules. *Information Science*, 177(7):1558–1566, 2007.

[GJ99] H. Gregersen and C. Jensen. Temporal Entity-Relationship Models-A Survey. *IEEE Knowledge and Data Engineering*, 11(3):464–497, 1999.

[HcW08] F. Heidenreich, I. Şavga, and C. Wende. On Controlled Visualisations in Software Product Line Engineering. In *Workshop on Visualisation in Software Product Line Engineering*, 2008.

[HKR$^+$07] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. An Algebraic View on the Semantics of Model Composition. In *European Conf. on Model Driven Architecture*, pages 99–113, 2007.

[HKW08] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Proc. of Intl. Conf. on Software Engineering*, pages 943–944. ACM, 2008.

[JF88] R. Johnson and B. Foote. Designing Reusable Classes. *Object-Oriented Programming*, 1(2), 1988.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. of Intl. Conf. on Software Engineering*, pages 311–320. ACM, 2008.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[Par03] J. Parsons. Effects of Local Versus Global Schema Diagrams on Verification and Communication in Conceptual Data Modeling. *J. Manage. Inf. Syst.*, 19(3):155–183, 2003.

[PGvdL05] K. Pohl, G.Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241, pages 419–443. Springer Verlag, 1997.

[Ras03] A. Rashid. A Framework for Customisable Schema Evolution in Object-Oriented Databases. In *Proc. Int'l Symposium on Data Engineering and Applications*, pages 342–346. IEEE Computer Society, 2003.

[SE06] M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering*, 11(3):174–193, 2006.

[SP94] S. Spaccapietra and C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Knowledgde and Data Engineering*, 6(2):258–274, 1994.

[SSEC07] M. Sabetzadeh, S.Nejati, S. Easterbrook, and M. Chechik. Consistency Checking of Conceptual Models via Model Merging. In *Proc. Int'l Conf. on Requirements Engineering*, pages 221–230. Springer-Verlag, 2007.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.