Building Information System Variants with Tailored Database Schemas Using Features

Martin Schäler[1], Thomas Leich[2], Marko Rosenmüller[1], and Gunter Saake[1]

[1] School of Computer Science, University of Magdeburg, Germany
{schaeler,rosenmueller,saake}@iti.cs.uni-magdeburg.de
[2] METOP Research Institute, Magdeburg, Germany
thomas.leich@metop.de

Abstract. Database schemas are an integral part of many information systems (IS). New software-engineering methods, such as *software product lines*, allow engineers to create a high number of different programs tailored to the customer needs from a common code base. Unfortunately, these engineering methods usually do not take the database schema into account. Particularly, a tailored client program requires a tailored database schema as well to form a consistent IS. In this paper, we show the challenges of tailoring relational database schemas in software product lines. Furthermore, we present an approach to treat the client and database part of an IS in the same way using a variable database schema. Additionally, we show the benefits and discuss disadvantages of the approach during the evolution of an industrial case study, covering a time span of more than a year.

Keywords: Tailoring DB schemas, feasibility study, software product lines.

1 Introduction

Databases (DB) are a integral part of many information systems (IS). In *software product lines* (SPL), software engineers aim at creating tailor-made software systems with the help of reusable artifacts [4,7]. An SPL forms a group of similar software systems sharing a set of identical and different functions. The reuse of high-quality artifacts in SPLs reduces the effort for maintenance and further development [6]. Although the use of SPLs for producing executable program code has been researched quite intensively, the impacts on data management and especially DB schemas are still fragmentary [10,16,21,23]. To generate a certain program (*variant*) of an SPL, a customer selects the *features* that he wants to include into his variant. A feature is a characteristic of a program that is relevant for some stakeholder [6]. Imagine a simple workflow management system that has an optional feature: *logging*. This feature is not part of every variant of the system, but can be chosen by a customer. Depending on the implementation of the *logging* feature, the DB schema needs additional relations containing the log data whenever a customer chooses this feature. Therefore, we need a possibility to *tailor the DB schema* according to the application's features when creating a new IS. Note, that simply designing a distinct schema for every potential variant manually is practically impossible, because the number of potential variants increases exponentially with the number of (optional) features. In addition, we argue that traditional engineering methods, such as using a global schema for all variants (see Section 3.2) raise the following challenges:

- Increased effort for maintenance and SPL evolution (e.g., integrate new features),
- Design limitations (e.g., no support for alternative features and table partitioning),
- Complex and thus hard to understand DB schema,
- Data integrity problems, because of missing integrity constraints in the DB schema,
- Solutions that do not scale, because of large number of variants.

In contrast to traditional engineering approaches, we aim at *generating* a *tailored* DB schema for every information-system variant of an SPL just as we tailor source code. To this end, we analyze whether the mechanisms used in SPL client programs, can be applied to the database schema to face the already mentioned problems. We extend previous work [21] suggesting to apply SPL techniques to ER modeling. This idea is adopted and extended to describe a holistic approach to model and generate an IS including client program and DB schema. Furthermore, we analyze the benefits and drawbacks of this approach. To the best of our knowledge, we provide the first empirical study that shows whether the generation of DB schema variants is possible for a specific data model and for an existing IS. Particularly, we investigate the following points:

1. Analyze why currently used approaches are not sufficient to face the previously mentioned issues,
2. Suggest an approach to tailor a DB schema to the client-program requirements and thus a new holistic approach for IS engineering,
3. Determine the *feasibility* of our approach for real-world applications and show *advantages* and *disadvantages* compared to existing strategies,
4. Investigate how the benefits and drawbacks *change over time*, when integrating new features and extending existing ones within the IS.

2 Software Product Lines Case Study

In the following, we introduce an SPL case study that we use to show the feasibility of our approach as well as to evaluate it's benefits and drawbacks.

2.1 The ViT-Manager

The ViT®-Manager is a family of industrial controlling tools for continuous improvement processes of the METOP Institute. A continuous improvement process defines a structured approach with regularly (continuous) meetings to identify and solve problems within a company, which increases productivity. Because the ViT-Manager is used commercially in several companies with different size, internal structure, and portfolio, the customers have a set of equal and differing requirements w.r.t. to one variant of the application. To face these challenges, the ViT Development Team refactored the application using a plug-in infrastructure to form an SPL. In the ViT Framework, a customer chooses additional features according to his needs, when generating a new variant of the

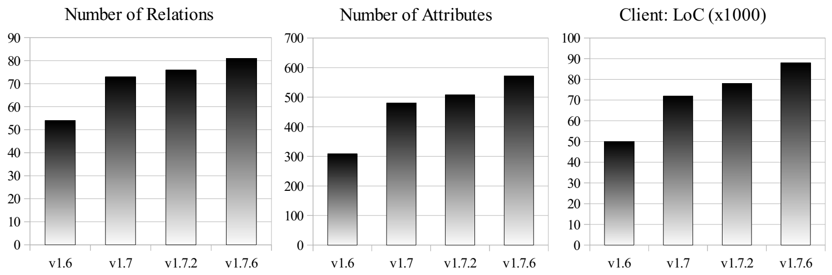


Fig. 1. Size of DB schema and client implementation for different versions of the ViT-Manager

application. While the client program was decomposed into *features* during the refactoring, the DB schema was not. Instead, a global DB schema for all variants was applied. We discuss disadvantages of this alternative solution in Section 3.2.

As we depict in Figure 1, the DB schema contained 53 relations with 309 attributes in version 1.6 (first version for which we implemented a variable schema, Feb. 2010). It increased to 81 relations including 572 attributes in version 1.7.6 (current version, Mar. 2011). Similarly, the size of the implementation on client side grew from about 50 thousand lines of code (KLoC) in version 1.6 to 88 KLoC in version 1.7.6. We consider this case study as appropriate to demonstrate our approach, because it is large enough to represent real-world challenges, while the evaluation complexity is manageable. Additional benefits are:

Productive use. The case study is used commercially. Hence, the danger of choosing a case study that does not include relevant problems is minimized. This improves the generalized results concluded from analyzing the case study.

Changes over time. Considering different versions of the case study (covering more than a year) allows us to extract results and conclusions about positive or negative effects of our approach during the evolution of the SPL case study. This includes adding new features to the SPL or extending and changing existing ones.

2.2 Feature Model of the Case Study

A *feature model* (FM) describes features and their relationships in a particular SPL [6]. Particularly, it defines, which feature combinations form a valid variant. Thus, an FM represents the variability of the SPL containing all valid feature combinations. In Figure 2, we depict the FM of the case study. There are four different types of feature relationships: mandatory, optional, alternative, and or-relationships. For instance, all variants of the case study have to contain a *Reporting* feature, because it is mandatory. In contrast, optional features do not have to be included. If a customer wishes to have a more detailed *Reporting* functionality, he chooses the optional *Management Cockpit*, which is then included into the generated program variant. Furthermore, one or more features connected via an *or*-relationship can be part of a particular variant. For instance, the *Login* of the case study can be performed via *LDAP*, *Local Authentication*, or both features may be included. Finally, the alternative-relationship defines that exactly one of the supported

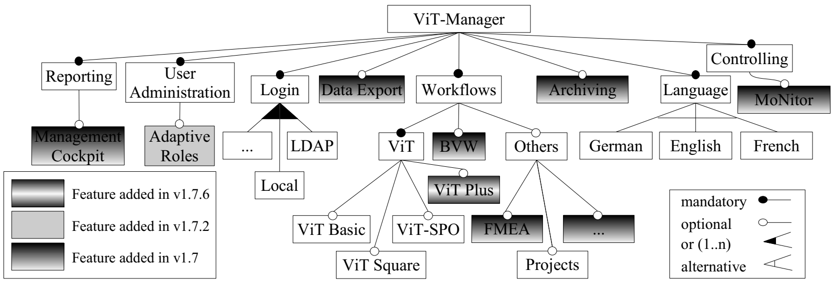


Fig. 2. Excerpt of the feature model of the case study

features can be included into one variant. For instance, to generate a variant of the case study a customer selects one of the supported language features (e.g., French). In Figure 2, we visualize the version history as well. Features such as *Reporting* and *Login* have been part of v1.6, whereas, the major release v1.7 contains several new *optional* features such as *Archiving* and *BVW*. In contrast, the minor releases v1.7.2 and v1.7.6 contain just a small number of new features and improvements of existing ones.

3 Problem Statement and State of the Art

Different users of an application have different requirements to its functionality. In client programs, these differences can be faced by using SPL techniques, such as components. While these techniques have been intensively studied for variability of executable program code, the variability at the DB schema level is still fragmentary [10,16,20,23]. In the following, we analyze different currently used approaches to model DB schemas in SPLs. Therefore, we define requirements that a tailored DB schema must fulfill. Note, that we can create thousands of valid variants of the case study. Thus, it is not possible to create every single tailored DB schema variant manually.

3.1 Requirements for Tailored DB Schemas in SPLs

To analyze different approaches, we consider several attributes of the process to create the single schema variants and of the resulting schema variants itself. We use two different criteria to describe the *modeling process* of a certain approach:

Modeling complexity. This criterion points out the complexity of creating the model of the (variable) DB Schema, which contains the basis for the DB schema variants.
Expressiveness of the model. In traditional modeling methods, the expressiveness of the model is limited. For instance, in a global schema, an engineer cannot integrate conflicting schema elements introduced by alternative features.

Evaluation of Schema Variants. The requirements for the single schema variants are:

Completeness. All DB schema elements (*relations and attributes*) necessary to perform the read and write operation in the single features included in this variant must be present in the variant's DB schema.

Complexity of schema variants. The size of the DB schema (number of *relations and attributes*) shall be reduced to a minimum. Particularly, there shall be no unused schema elements. We argue, that this improves the understandability of the schema variants, which is important for maintenance and further development.

Data integrity. All integrity constraints have to be included for the schema elements in a specific schema variant, to guarantee consistent data in the IS. This includes *primary and foreign keys*, *attribute value domains*, and *not null* as well as *check* constraints.

3.2 Limitations of Currently Used Approaches

Subsequently, we discuss three traditionally used approaches and describe which problems arise when applying them for SPL development. This discussion motivates the necessity for a new approach as well as it serves as basis when analyzing benefits and drawbacks of our new approach in Section 5.

Global DB Schema. Often, for every variant of an SPL the same global schema is applied [21]. This schema contains every schema element that is used at least in one variant. Thus, the schema contains every schema element that is necessary (*completeness*). On the other hand, major parts of the global schema can be unused in this particular variant. Consequently, the highly complex schema is unnecessarily hard to understand, which complicates maintenance and evolution of the SPL. Additionally, unused parts can impose integrity problems. Imagine an attribute used only in one variant. When it is used it shall be not null, or some check constraint is defined on it. Because it is part of *every* variant, you cannot define *integrity* constraints at the schema level, but on client-program side, which can lead to inconsistent data (e.g., by programming errors). Furthermore, the global schema does not exist in every case. As a result, conflicting schema elements introduced from alternative features cannot be defined in a global DB schema (expressiveness). To circumvent this problem, YE et al. discuss that overloading schema elements (i.e. using the same schema element for different purposes instead of renaming it) is possible, but causes highly confusing DB schemas [23]. Using a global schema can be seen as the standard approach, hence the modeling complexity is used as reference for all other approaches [21].

View-Based Approaches. View-based approaches [5] generate views on top of the global schema that emulate a schema variant for the client, which may be seen as an annotative approach [13]. Thus, the global schema is still part of every DB schema variant, the approach inherits the problems of the global schema. Unfortunately, the variant's schema complexity does even increase, because the additional schema elements for the view, emulating the variant, have to be included as well. Furthermore, there is additional effort to generate views when modeling the DB schema. This approach has benefits in data integrity, because the views emulating the schema variants can contain additional integrity constraints, which cannot be included into the global schema. Thus, the expressiveness of the model is also better than in the global schema approach.

Framework Solutions. In a framework approach, the plugins implement the features of the SPL and therefore this approach is a form of physical decomposition [13]. A plugin contains additional program code and an own DB schema. The DB schema variant is build from the single plugins that add the additionally required schema elements [21]. Thus, it fulfills the *completeness* requirement. Consequently, it also contains only schema elements that are needed in this variant. Unfortunately, using frameworks could lead to table partitioning, when two or more plugins use the same real world entity. Hence, this has negative impact on the complexity of the schema variants and limits the modeling expressiveness. Furthermore, consistency checking is implemented on client side, because there are no intra-plugin integrity constraints, which can lead to data integrity problems. The effort to model the DB schema is higher than modeling a global schema, because you have to take care of additional challenges such as naming conflicts, which are usually solved by naming conventions.

4 Solution - Holistic Approach for Client and DB Schema

This section contains the basic overall idea of our approach and explains how to obtain the variable DB schema. Therefore, we describe the relationship between features and DB-schema elements. Furthermore, we present a semi-automatic approach to relate schema elements to features.

Basic Idea. To create a tailored variant of the IS variant, a customer first selects the desired features (see Figure 3(a)). Every feature contains one model for its functions in the client program and another one for the DB schema (Figure 3(b)). Furthermore, each of the two models of a particular feature points to different implementation artifacts (e.g., source code files). Finally, a well defined composition strategy (see Section 4.3) creates the variant including the tailored client program and DB schema.

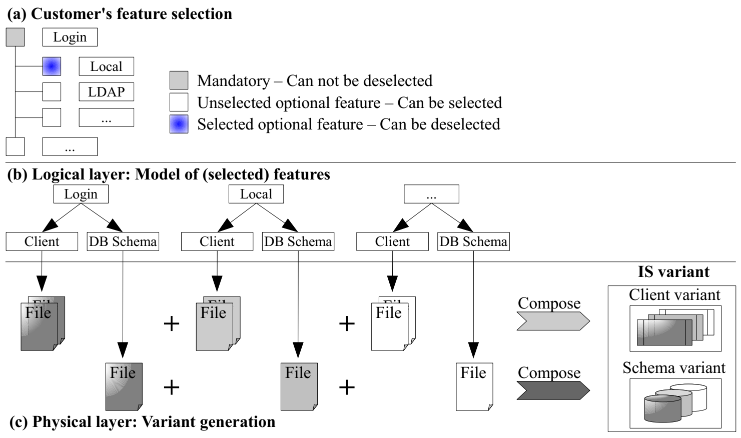

(a) Customer's feature selection
Login
Local
LDAP
...
...
Mandatory – Can not be deselected
Unselected optional feature – Can be selected
Selected optional feature – Can be deselected
(b) Logical layer: Model of (selected) features
Login
Local
...
Client
DB Schema
File
+
Compose
IS variant
Client variant
Schema variant
(c) Physical layer: Variant generation

Fig. 3. Basic idea to create a particular variant

4.1 Relationship between Features and DB Schema Elements

As mentioned in Section 2, a feature is some characteristic that is important for some stakeholder. Furthermore, the features are the units a customer selects to create a tailored variant, which includes a DB schema. Therefore, we have to define the relationship between features and DB schema elements to model the *variable* DB schema. Moreover, we need to discuss the interaction of features on client and data-base-schema level. A feature at DB schema level contains all schema elements (relations, attributes, and integrity constraints) that this feature on client side needs to perform the read and write operations within its specific source code on client side. Thus, we tailor the DB schema w.r.t. the requirements of the feature on client side. As a result, there is one feature model for the whole IS allowing us to easily generate the single variants of the IS (see Section 4.3).

Note that the relationship between a feature on client and DB schema level ensures *completeness* and minimizes the *complexity* of the DB schema variant (see 3.1). Every schema element that a selected feature needs in a variant, is contained in the feature and therefore added to the schema variant during composition. Furthermore, the schema variant contains no unused schema elements for the same reason. Alternatively, the feature models could contain only the additionally required schema elements. This approach has the benefit that schema elements cannot be mapped to multiple features. This can lead to conflicting definitions of a schema element in *one* variant during the evolution of the SPL. The drawback of this alternative is, that we have to decide to what feature a schema element belongs, especially when they are only required by optional features. However, a high number of redundancy suggests a refactoring of the client programs source code.

Mapping Features to DB Schema Elements. Previously, we defined that a feature in the variable DB schema is a mapping of schema elements to this feature. Furthermore, we show how to create this mapping of schema elements to features representing the variable schema. To create the model of a variable DB schema there are two different alternatives:

1. The single features can be extracted from a previously used global DB schema.
2. Especially when creating the SPL from scratch, features on DB schema level are modeled in features instead of views, which have to be composed to form a schema variant.

Extracting the features from a global schema has the advantages that there is already a consistent DB schema. The main task in this case is mapping schema elements to features in a reasonable way. In contrast, modeling the features without a consistent schema, a development team faces additional challenges (e.g., naming conflicts) widely known from view integration [3].

Currently, SPL techniques are often used when traditionally methods are reaching there limitations [6]. That means that previously monolithic software is refactored into an SPL such as the case study. Thus, in these cases there also is a global schema and therefore we concentrate on this alternative.

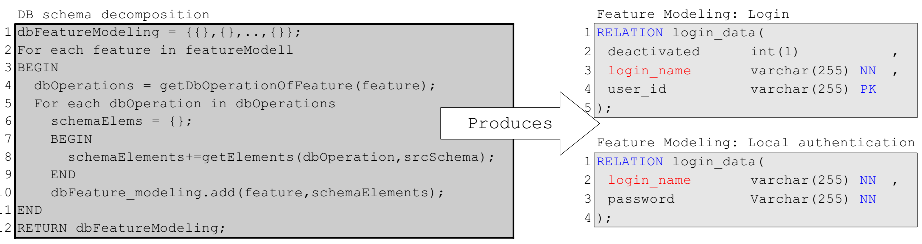
4.2 From a Global DB Schema to a Variable Schema

Subsequently, we explain a semi automatic schema decomposition to map schema elements to features based on the client implementation. Therefore, we assume that a global previously used DB schema exists, because this is the standard case as we discussed in Section 3.2. The important point is that the mapping fulfills the *completeness* requirement recently defined.

Preconditions. The schema decomposition needs two preconditions: (1) Previously used global DB schema and (2) Client implementation that is separated into features. First, the decomposition needs a previously used global schema, because we need the schema elements, which includes element names, value domains (attributes), and integrity constraints defined for each particular element. The second precondition is that the client implementation is separated into features, which is true in SPLs. Consequently, the decomposition can identify all necessary schema elements (including integrity constraints) for a particular feature as postulated in the relationship of a feature on client and DB schema level.

Semi Automatic DB Schema Decomposition. The decomposition procedure determines the mapping of schema elements to every feature in the FM. First, for every feature in the FM the decomposition identifies the database operations (e.g., select and insert queries) in the source code. Second, for all of these operations it identifies the required schema elements and adds them to the feature's DB schema model. For instance in *Select* queries, the decomposition adds attributes used in the *select* and *where* part as well as the relations named in the *from* part to the model of the particular feature. Moreover, when adding a schema element to the feature, additional information extracted from the global schema, are added to the model, such as an attribute value domain and integrity constraints. In Figure 4, we show the result of the schema decomposition for two features of the case study. Each feature contains one relation (usually a feature contains several relations) and a specific number of attributes in a syntax form that is closely related to the SQL-Standard. Note, that integrity constraints, such as primary keys (PK) and not null constraints (NN) can be added to an attribute.

Problem Referential Integrity. As we presume that there is a previously used (consistent) global schema, there are no problems with integrity constraints (e.g., primary keys). By contrast, foreign keys are problematic if they reference from or to a schema element not existing in a particular variant. To avoid this problem, we propose rules that ensure that the referenced schema elements are present in the variant. The rules representing valid introductions of foreign key within a feature are: (1) The schema elements are located within the feature itself, (2) the referenced element is part of a parent feature, or (3) the referenced element is part of a feature the current feature has a *requires* relationship to. As a result, it is generally not possible to define foreign keys between optional features. For instance, in Figure 2 it is not possible to define a foreign key between the features *ViT Basic* and *ViT Square*. Whereas, it is possible to define foreign key in feature *Projects* to schema elements of feature *Others*, because *Others* is a parent feature (rule 2). If one really wants to have foreign keys between optional



DB schema decomposition

```
dbFeatureModeling = {{},{},..,{}};
For each feature in featureModell
BEGIN
  dbOperations = getDbOperationOfFeature(feature);
  For each dbOperation in dbOperations
    schemaElems = {};
    BEGIN
      schemaElements+=getElements(dbOperation,srcSchema);
    END
    dbFeature_modeling.add(feature,schemaElements);
END
RETURN dbFeatureModeling;
```

Feature Modeling: Login

```
RELATION login_data(
 deactivated       int(1)           ,
 login_name        varchar(255) NN  ,
 user_id           varchar(255) PK
);
```

Feature Modeling: Local authentication

```
RELATION login_data(
 login_name        varchar(255) NN  ,
 password          Varchar(255) NN
);
```

Fig. 4. Listing: DB schema decomposition

features, you have to define a *requires* constraint in the feature model (rule 3) making this dependency explicit. This ensures that only valid variants of the IS can be created.

Current Limitations of the Schema Decomposition Approach. Currently, the identification of all DB operations within the feature specific source code is not trivial in all cases, especially when DB operations are assembled over multiple lines or depend on user interactions. For the same reasons, the identification of necessary schema elements in a query is not trivial. Moreover, when using a global schema not all integrity constraints are defined on the DB schema level (see Section 3.2), but rather on the implementation level. In the case study, these challenges did not occur because of previously introduced programming conventions, which are common in todays programming. Therefore, we adjourn these challenges to future work. Note, that similar challenges exist in the decomposition of program code [4].

4.3 Composing a Tailored Information-System Variant

We already defined that the model on DB side contains a mapping of schema elements to features. Furthermore, we emphasized that the feature on DB schema level contains all necessary elements for the feature on client side to perform the read and write operations of this feature. Subsequently, we present a mechanism to compose tailored IS variants containing the client program as well as the database schema.

Superimposition Composition Mechanism. As noted in Figure 3, the approach needs a well defined composition strategy to generate a tailored IS variant. For this reason we use *superimposition*. Superimposition is a language-independent composition strategy, that has been applied successfully in software engineering [2] and also in view integration [3]. Thus, the composition is able to handle the client implementation and the corresponding DB schema. In [1], APEL et al. present Feature House, which offers a general support for composing software artifacts. It has been tested for several case studies on client side, written in languages such as Java, C, or Haskell. The composition operation is denoted by the •-operator [2].

$$\bullet : F \times F \rightarrow F \qquad\qquad V = f_n \bullet \ldots \bullet f_2 \bullet f_1 \qquad (1)$$

The •-operator merges the (sub) structures of two features (F) to create a new feature that contains all concepts of both input features without duplicates. For instance when superimposing DB schemas, the input consists of several DB schemas (model of features at DB side) and the result is a DB schema containing all schema elements of the input schemas, without any duplicate relations or attributes. To create a schema variant (V), the DB schemas of all selected features (f_i) are superimposed.

Structure of Features at Implementation Level. The •-operator works on *feature structure trees* (FST), which represent the internal structure of the implementation of a feature. They can be seen as simplification of an abstract syntax tree [2]. In Figure 5, we show the FST of the already known feature *Login* and *Local Authentication* of the case study on DB schema level. The FST can be generated quite intuitively from the model as shown in Figure 4. The result of the composition contains all attributes of the input features with the desired integrity constraints. FST nodes of two features that share the same path from the root node (e.g., attribute *login_name*) are merged and thus show only once in the result as intended.

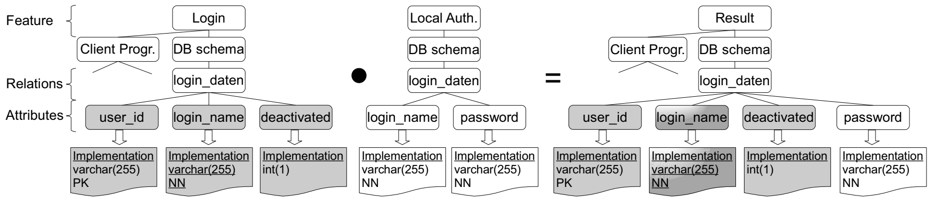


Fig. 5. Composition of feature structure trees (FST)

5 Evaluation

In the following, we apply the previously presented approach to the case study and evaluate the results. Therefore, we first analyze the feasibility of our approach before discussing its impacts on maintenance, data integrity, and further development. Finally, we evaluate the overall advantages and drawbacks of our approach compared to the traditionally used ones from Section 3.2. To ensure validity of the evaluation, we rely on interviews of the ViT Development Team, which is experienced in applying global DB schemas to SPLs, because the case study previously used a global schema. Furthermore, this team tested the variable schema approach for more than a year. Finally, we strengthen our conclusions with specific measurements if possible.

5.1 Feasibility of the Approach

We want to evaluate whether the variable DB schema approach is manageable in practice. Thus we also have to evaluate the modeling process, the result of the modeling (features), and the variant generation process. As stated in Section 4, the modeling process, which includes the decomposition of the global schema into features is laborious. Furthermore, the decomposition procedure is not generally automatable, because of open research challenges and thus needs manual recall. However, according to interviews the modeling is still manageable. Moreover, the modeling effort is an investment for the automated generation of a tailored DB schema variant. A customer simply chooses the desired features and the DB schema (and the client) are generated automatically. Additionally, the size of the features scales in the case study (see Figure 1), because the size on client and on DB side seems to correlate. This effect does not change significantly over time. As visualized in Table 1, even when comparing two major releases such as v1.6 and v1.7 the correlation exists.

Table 1. Correlation of size on DB and client side

Features v1.6	Rank Size DB/client	Attributes	Relations
ViT	1 / 1	88	16
ViT-SPO	2 / 2	80	14
BVW	4 / 3	75	12
ViT Square	3 / 4	65	12
ViT Basic	5 / 5	63	12
...			
LDAP	24 / 21	1	1

Features v1.7	Rank Size DB/client	Attributes	Relations
Archiving	1 / 4	124	17
ViT-SPO	2 / 1	105	16
ViT	3 / 2	88	16
BVW	4 / 5	75	12
ViT Square	5 / 3	66	12
...			
LDAP	30 / 26	1	1

Additional Challenges Due to Feature Interactions. We identified two additional challenges of the variable schema approach: Redundantly defined schema elements and changing feature models. Redundant schema elements increase the size of a feature on database side. Thus, the features were extracted from global schema, two features that can be present in one variant cannot contain different models of the same schema element, which would produce an error during composition. But, when modeling the features without a previously used schema this challenge has to be faced, which rises also in view integration. The second challenge results because of the combination of optional features. The schema variant might need additional schema elements when a customer chooses two optional features to generate a variant. These schema elements are not part of the schema variant when only one of them is chosen. This effect is also known on client side (e.g., glue code), but to the best of our knowledge it has not been identified at DB schema level.

In the case study, there is an optional *Archiving* feature that archives data of different *optional* features, such as *ViT basic* or *BVW*. Thus, these optional features are not part of every variant. Therefore, the archiving feature does not need the tables to archive the *BVW* data if the *BVW* feature is not included into the variant. To minimize the complexity of the schema variant, these unneeded relations should not be included. Hence, we move the additionally needed schema elements into *derivatives*, which are included automatically when both optional features are part of the variant [15]. Particularly, the composition mechanism includes *Archiving:BVW* (Figure 6) into the schema variant, when a customer selects *BVW* and *Archiving*. Using Derivatives raises the expressiveness of the model and decreases the size of the *Archiving* feature in some variants dramatically, by making the modeling process more complex.

Trade-Off. According to our interviews, the trade-off between the drawbacks of our approach, such as additional modeling overhead, and the advantages like automated variant generation are beneficial only when handling multiple variants of an application. But the specific costs, such as modeling the variable schema, related to the amount

Features	Relations	Attributes
Archiving:BVW	12	59
Archiving:ViT SPO	7	32
Archiving:ViT Square	3	16
Archiving:ViT Basic	4	13
Archiving:Others	1	8

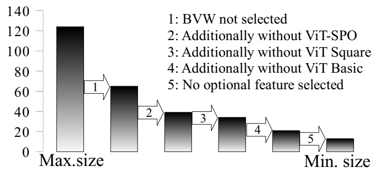


Fig. 6. Derivatives: Size of the Archiving features in different feature combinations

of time saved (e.g., for manually tailoring a variant or additional bug fixing if using a global schema) to the customers needs, is imprecise for the following reasons. The complete costs for alternative solutions are unknown, because these approaches have not been implemented and the previous costs can hardly be taken into account because the tasks are highly different. Moreover, there a lot of fine-grained costs where the specific amount of time is unknown. For instance for bug fixing in the variable-schema approach, we do not know the amount of time, as only the time from reporting the bug until fixing it is known. After all, the numbers we have and interviews with the developers suggest that for the *ViT-Manager* the variable-schema approach is beneficial when there are at least five (different) variants used productively.

5.2 Improving Maintenance, Data Integrity, and Future Development

Improving the *maintenance* has two aspect. First, the *effort estimation* to maintain the DB schema or the data inside shall be improved. When using a global schema, it is complex and thus hard to understand. Hence, the effort estimation is often imprecise, because of possible side effects. Furthermore, the maintenance process itself shall be supported in a positive way. This issue is highly related to reducing the *variants schema complexity*. Here we argue that not having unnecessary schema elements in the variant schema is a suitable way to reduce its complexity. The effort to maintain a schema variant becomes more predictable, because we have a mapping of a particular function to schema elements (features). Thus, the estimation of possible side effects becomes easier.

The complexity of a schema variant is minimized, because it contains only necessary schema elements. Therefore, the understandability of the DB schema variants rises and is additionally supported by the mapping of schema elements to features. When comparing the complexity reduction of the schema variants in different versions (Figure7), the benefits of the variable schema become even more obvious for the case study. The minimum configuration (no optional feature selected) of v1.6 covered about the half of the maximum configurations schema elements (all optional features selected). Whereas, the difference in v1.7.6 is about one fourth. Furthermore, the DB schema of the minimum configuration remains nearly stable. This is, because many new *optional* features have been added to the case study, but the core functions remained roughly unchanged, which is common in SPLs. Thus, we conclude that the benefit of our approach increases over time, when new optional features are added to the SPL.

According to our interviews, after introducing the variable schema the number of support request slightly increased for a quarter of a year, especially because of problems related to the db schema. After this, there was significant decrease of support request even if introducing new features. We hypothesize that one of reason therefore is the variable schema, but we cannot say to what extend.

Strong Benefit: Improving Data Integrity. One of the strongest point supporting the variable schema approach, is the benefit w.r.t. data integrity. Using the variable schema allows to reintroduce the integrity constraints that had to be dropped (see Section 3.2), because of the global schema approach. Additionally, integrity constraints that were encoded on client side, can now be added to the DB schema. As a result the variable schema approach *ensures integrity* of the data and is therefore beneficial here. Furthermore, tricks as inserting dummy values to circumvent Not Null constraint for attributes not used in this variant are not necessary any more. Therefore, the variable schema approach is beneficial for *data quality* as well. Moreover, the rules from Section 4.2 guarantee that we can only introduce valid integrity constraints and that additional restrictions of the variability of the SPL (introduced by foreign keys referencing from an optional feature to a different optional feature) are visible in the Feature Model.

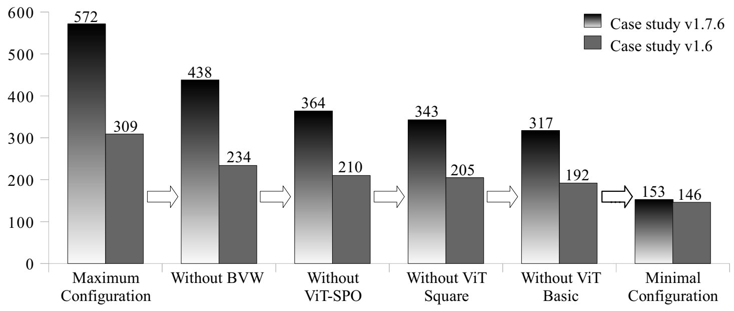


Fig. 7. Complexity reduction of the schema variants in different versions of the case study

Further Development. Extending the case study with new features becomes easier and is more predictable in the variable DB schema approach. First, a software engineer can simply add new features to the composition process. Furthermore, the mapping between features and schema elements is helpful when designing new features or extending existing ones. This was highly useful, when designing the *Archiving* feature and its *derivatives*, because for every feature was know, which schema elements are necessary and thus have to be archived. Challenges arise when modifying schema elements used in different features in preserving the model's consistency. Additionally, we see naming problems when adding new features, which also arise in view integration. However, we are confident, that this problem can be solved with adequate tool support.

5.3 Comparison to Existing Approaches

In Table 2, we resketch the result of currently used approaches of Section 3 and compare them to the variable schema approach. The scoring of variable schema is based on the discussion in Section 3.2. The variable schema has strong benefits due to the complexity reduction of schema variants and the improvement of data integrity (also helping to improve data quality). Furthermore, the model expressiveness increases, which allows conflicting schema elements and changing feature models that is not possible with a global schema approach. By contrast, the global schema approach and frameworks have benefits w.r.t. modeling complexity. The variable schema needs a previously used global schema, which has to be decomposed into features. Alternatively, the modeling of features instead of views, including all derivatives is also more complex than modeling one global schema. This is the trade-off, our approach needs to improve maintenance and further development.

Table 2. Evaluation of the variable schema approach compared to currently used approaches

	Modeling		Schema Variants		
	Complexity	Expressiveness	Completeness	Data Integrity	Schema Complexity
Global DB Schema*	+	--	++	-	-
View Approach*	--	-	++	+/-	--
Framework Solutions	+/-	+/-	++	-	+
Variable Schema	-	+	++	++	++

++ very good, + good, 0 neutral, - unhandy, -- very unhandy *Approach only possible if global schema exists.

6 Related Work

There has been little work published in literature that deals with tailoring DB schema in SPLs or similar concepts. In [23], *Ye* et al. discover the need for tailoring software systems at different levels and of different granularity, which includes the DB schema. In their analysis of an industrial case study, they discover the use of schema element overloading in a global schema, which creates a hard to maintain DB schema. Furthermore, service engineers add additionally required schema elements manually. However, this approach is only possible in case studies with very little variability at DB schema level.

In his PhD Thesis [16], MAHNKE proposed a component approach that is similar to the framework solution described in Section 3.2. Unfortunately, his ABLE-SQL requires an extension of the SQL:1999 standard, which is not necessary in our variable schema approach. Whereas view-based approaches, such as [18,22], use view integration strategies or view tailoring [5]. In Section 3.2, we also discovered limitations of these approaches because they need a global schema and build views on top of the global schema. Thus, the complexity of the schema variants even increases.

Recently, DYRESON et al. [10] presented an aspect-oriented approach to integrate optional features into XML, which the authors adopted to the relational data model in [9]. This approach needs a change of the query processing to evaluate the aspects, which is not necessary using the variable schema, because our approach is totally transparent to the underlying DB system.

In [17], RASHID identified the lack of variability, but concentrates on evolutionary changes of the DB schema in his aspect-oriented framework approach and not on composing different tailor-made schema variants. Furthermore, model-merging approaches suggest more complex mechanism (e.g., Comparison phase to identify equivalent elements) than our approach and are thus more expressive [12,8,19]. However, superimposing Feature Structure Trees is less complex and sufficient for our purposes.

Finally, our approach uses composition (physical separation of concerns) [13] as the schema elements of each feature are kept in their own file (refer to Figure 3) and need to be composed to form a particular variant. Alternatively, it may be possible to use an annotative approach (virtual separation of concerns) as there are solutions such as CIDE [14] and the concept generally is language independent as well. However, as we needed to integrate new features during the evolution of the case study, we applied the compositional approach. As a result, we can simply integrate new features into the composition process or delete existing ones without changing respective annotations in the virtually separated schema.

7 Conclusion and Future Perspectives

In this paper, we have shown the feasibility of building IS variants with tailored DB schemas in SPLs using a variable schema. First, we analyzed the limits of existing approaches. Second, we discussed the relationship of features on client and database side. Third we showed how to model a variable schema and use the superimposition composition mechanism that generates a particular IS variant having a tailored schema automatically. The approach has been applied to a case study to show advantages and disadvantages compared to existing approaches and its influence over time. The variable schema has significant advantages, such as reducing the complexity of DB schema variants reduction and increased expressiveness. Whereas, the modeling is much more complex compared to existing approaches.

In future, we have to solve several challenges when decomposing a previously used DB schema into features. Furthermore, a better tool support is highly needed to apply our approach to large-scale case studies and to verify the results conducted from the ViT-Manager case study. Thus, we want to integrate our approach into existing solutions (e.g., [1]) to tailor source code or to visualize the relationship between features and respective implementation artifacts such as [11]. As result, we will furthermore support the holistic design of IS. Better tool support additionally includes research for multi-lingual variability (e.g., SQL and Java). Finally, we will analyze how to upgrade one variant to a different one efficiently (i.e., schema and data). Especially we will analyze, which benefits and (new) challenges arise in DB schema evolution.

Acknowlegements. This paper has been funded in part by the German Federal Ministry of Education and Science (BMBF) through the Research Program under Contract FKZ: 13N10818 and 13N10819. We would also thank Martin Kuhlemann and Sandro Schulze for their support for this paper.

References

1. Apel, S., Kästner, C., Lengauer, C.: Featurehouse: Language-independent, automated software composition. In: Proc. Int'l Conf. on Software Engineering, pp. 221–231. IEEE (2009)

2. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 20–35. Springer, Heidelberg (2008)
3. Batini, C., Lenzerini, M., Navathe, S.: A comparative analysis of methodologies for database schema integration. ACM Computing Surveys 18, 323–364 (1986)
4. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
5. Bolchini, C., Quintarelli, E., Rossato, R.: Relational Data Tailoring Through View Composition. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 149–164. Springer, Heidelberg (2007)
6. Clements, P., Northrop, L.: Software product lines. Addison-Wesley (2001)
7. Czarnecki, K., Eisenecker, U.: Generative programming: Methods, tools, and applications. Addison-Wesley (2000)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
9. Dyreson, C., Florez, O.: Data aspects in a relational database. In: Proc. Int'l Conf. on Information and Knowledge Management, pp. 1373–1376. ACM (2010)
10. Dyreson, C., Snodgrass, R., Currim, F., Currim, S., Joshi, S.: Weaving temporal and reliability aspects into a schema tapestry. Data Knowl. Eng. 63, 752–773 (2007)
11. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: Comp. Proc. Int'l. Conf. on Software Engineering, pp. 943–944. ACM (2008)
12. Jossic, A., et al.: Model integration with model weaving: a case study in system architecture. In: Proc. Int'l. Conf. Systems Engineering and Modeling, pp. 79–84. IEEE (2007)
13. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proc. Int'l Conf. on Software Engineering, pp. 311–320. ACM (2008)
14. Kästner, C.: Cide: Decomposing legacy applications into features. In: Demonstration at Proc. Int'l. Software Product Line Conf., pp. 149–150 (2007)
15. Liu, J., Batory, D., Lengauer, C.: Feature-oriented refactoring of legacy applications. In: Proc. Int'l Conf. on Software Engineering, pp. 112–121. ACM (2006)
16. Mahnke, W.: Towards a modular, object-relational schema design. In: Doctoral Consortium at Proc. Int'l. Advanced Information Systems Engineering, pp. 61–71. Springer (2002)
17. Rashid, A.: A framework for customisable schema evolution in object-oriented databases. In: Proc. Int'l. Symp. on Database Engineering and Applications, pp. 342–346. IEEE (2003)
18. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. 11(3), 174–193 (2006)
19. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: Proc. Int'l Conf. on Requirements Engineering, pp. 221–230. Springer, Heidelberg (2007)
20. Schäler, M., Leich, T., Siegmund, N., Kästner, C., Saake, G.: Generierung maßgeschneiderter Relationenschemata in Softwareproduktlinien mittels Superimposition. In: Proc. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, pp. 414–534. GI (2011)
21. Siegmund, N., Kästner, C., Rosenmüller, M., Heidenreich, F., Apel, S., Saake, G.: Bridging the gap between variability in client application and database schema. In: Proc. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, pp. 297–306. GI (2009)
22. Spaccapietra, S., Parent, C.: View integration: A step forward in solving structural conflicts. IEEE Trans. on Knowl. and Data Eng. 6(2), 258–274 (1994)
23. Ye, P., Peng, X., Xue, Y., Jarzabek, S.: A case study of variation mechanism in an industrial product line. In: Proc. Int'l Conf. on Software Reuse, pp. 126–136. Springer (2009)