Managing the Evolution and Customization of Database Schemas in Information System Ecosystems

Hendrik Brummermann[1], Markus Keunecke[2], and Klaus Schmid[2]

[1] Hochschul-Information-System GmbH, Hanover, Germany
brummermann@sse.uni-hildesheim.de
[2] Institute of Computer Science, University of Hildesheim, Germany
{keunecke,schmid}@sse.uni-hildesheim.de

Abstract. We present an approach that supports the customization and evolution of a database schema in a software ecosystem context. The approach allows for the creation of customized database schemas according to selected, supported feature packs and can be used in an ecosystem context, where third-party providers and customers augment the system with their own capabilities.

The creation of the final database schema is automatic and also the relevant updates of individual feature packs can be automatically handled by the system.

Keywords: Database Schema, Evolution, Customization, Feature Packs.

1 Introduction

Information systems play an important role in the management of organizations. But in order to be effective, they often need to cover a wide range of functionality to support all relevant user roles effectively. For example an information system for a university needs to address personnel and finances, but also lectures, exams, students, etc.

Today software ecosystems are an important topic due to their business implications. Often in an ecosystem, a core development organization provides a platform, which third party vendors extend with additional functionality. This allows customers to integrate features from different vendors into their systems to create a product that fits their needs [9]. This concept can be applied to software product lines [2] and information system product lines [4].

In order to allow such extensions, features need to be encapsulated into, what we call, feature packs, which may consist of both an implementation and a partial data-model. When feature packs are installed, updated and removed by customers, the database structure and content must be adjusted accordingly. In this paper, we will describe an approach that supports this and evaluate it based on a number of case studies.

Our approach is based on a relational database and object/relational mapping. While our case study is an information system in the university management domain and implemented in Java, our approach is not specific to any particular programming language or domain. It can be used in other information system ecosystems.

The remainder of this paper is structured as follows: Section 2 describes the case study context and Section 3 defines the general problem. In Section 4 we describe our solution approach, which we evaluate in Section 5. In Section 6 related work is discussed before we conclude the paper in Section 7.

2 Case Study

Our research is motivated by and based on experiences with a large-scale information system development. The approach we discuss in this article was driven by needs from this context and integrates parts that are already in use for several years as well as parts that have been created to address recent and upcoming needs. In this section, we will describe the case study context. In particular, we will introduce some features, which we will use as examples throughout this paper.

2.1 Information System HISinOne

The HISinOne-system is the basis for our case study. We describe it and its evolution in this section. The producer Hochschul-Informations-System GmbH (HIS) is market leader in Germany for university management systems for 40 years. The HISinOne-product covers many areas that are relevant to universities like management of students, personnel, or lectures. It is a large information system under active development. Table 1 shows its growth in terms of lines of code, database tables and columns, and foreign-key constraints over recent years. HISinOne uses O/R-mapping with rather small objects. This results in small tables with an average of less than eight columns per table.

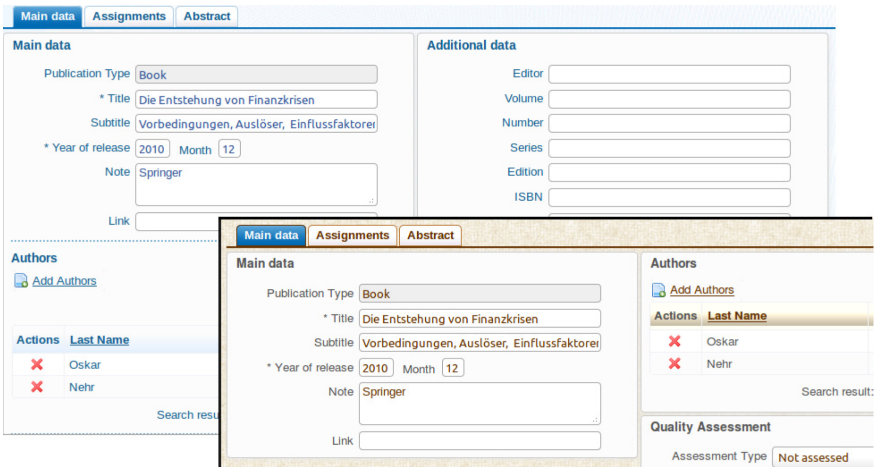
HISinOne must be highly customizable due to diverse requirements originating from a number of sources. Laws regarding universities are defined by each of the 16 federal states in Germany. There are different sizes and categories of universities (e.g., universities focusing on research, universities of applied science, and universities of sports or music). Further, even in a single university there exists diverse regulations, e.g., concerning examinations, that the system must handle simultaneously. An approach for handling the system configuration in this context was described in [4]. Customizations range from small modifications like fitting the look-and-feel to the corporate design over business processes adaptations to replacing complete functional areas like lecture management with products from other vendors. Figure 1 compares two different HISinOne Publication Management products, which differ in the number of features and in customizations of the user interface. These variations can be very significant as a previous study [11] showed.

HISinOne is at the core of its own information system *ecosystem* as customers and third-party vendors develop their own customizations and extensions as feature-packs. These extensions may consist of both program logic and additions to the database. Customers may install feature-packs by themselves at any time without support by HIS.

HIS releases about two major versions per year and additional service packs as required. Customers, however, face the problem that it is difficult to identify a good

Table 1. Metrics of the last three versions of HISinOne

version	code-lines	tables	columns	foreign-keys
2010	3 300 000	524	3816	938
2011	4 900 000	718	5638	1272
2012	5 300 000	814	6362	1434



Main data | Assignments | Abstract

Main data

Publication Type	Book
* Title	Die Entstehung von Finanzkrisen
Subtitle	Vorbedingungen, Auslöser, Einflussfaktorer
* Year of release	2010 Month 12
Note	Springer
Link	

Additional data

Editor	
Volume	
Number	
Series	
Edition	
ISBN	

Authors

Add Authors

Actions	Last Name
✗	Oskar
✗	Nehr

Search resu

Main data | Assignments | Abstract

Main data

Publication Type	Book
* Title	Die Entstehung von Finanzkrisen
Subtitle	Vorbedingungen, Auslöser, Einflussfaktorer
* Year of release	2010 Month 12
Note	Springer
Link	

Authors

Add Authors

Actions	Last Name
✗	Oskar
✗	Nehr

Search result:

Quality Assessment

Assessment Type	Not assessed

Fig. 1. Screenshot of two different HISinOne Publication Management products

time-frame for updating the complete installation as at any time some part is under intensive use. Thus, it is desirable to support the *evolution* of individual feature packs without ramifications for other parts of the installation. Possible issues, which may arise in update situations, are described in [3].

2.2 Features Used as Examples in This Paper

In this section, we will introduce the research management domain of HISinOne and relevant feature packs as a reference example for the rest of this paper.

The Research Management domain consists of features for the management of research projects and publications as shown in Figure 2. The core *Publication Management* may be extended with the following features:

Research Output Measurement (F1): Research projects document results by creating publications, therefore this feature links projects and publications to enable output measurements.

Proceedings Publication Type (F2): Support for the additional publication types Proceedings and InProceedings, which are not supported by the core system.

Publication Quality Assessment (F3): The possibility to assess the quality of publications like workshop, conference, journal.

While these three features may be developed independently (even by third party vendors or customer), they interact with each other at runtime: end users expect a single dialog for publications, where they can enter all relevant data, without considering the structure of the system. Also, the additional publication types provided by the *proceedings publication* feature impact the behavior of the *quality assessment* and the *output measurement* features.

The ecosystem and evolution requirements of our context result in challenges that are relevant to other information system ecosystems, too. We discuss them below.

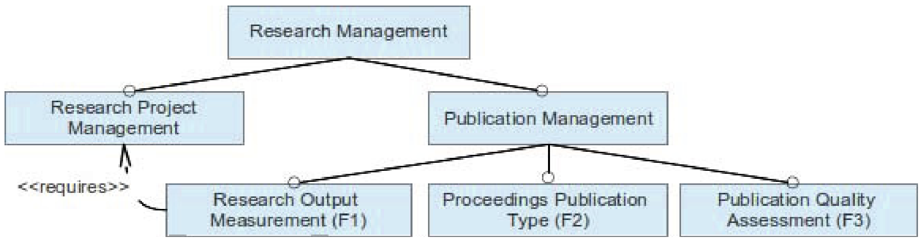

Research Management
Research Project Management
Publication Management
<<requires>>
Research Output Measurement (F1)
Proceedings Publication Type (F2)
Publication Quality Assessment (F3)

Fig. 2. Variability model of the publication management features

3 Problem Statement

While our research is motivated by problems, which occur in practice in a specific company as described in Section 2, the issues are of broader relevance to information system ecosystems. We distinguish two problem areas: in an ecosystem a global data schema can no longer be enforced, rather it needs to be broken down into feature-specific fragments. Further, when partial updates are done the database must be appropriately updated without a negative impact on existing data.

The identified problems areas lead to the following research questions:

RQ1 How can separate organization-wide data-models in small-scale feature-specific models and manage them on this level?
RQ2 How can features, developed by customers and third party vendors, be integrated into one combined data-model?
RQ3 How can the database be modified at the customer site to fulfill the requirements of installed and updated feature packs?

The following constraints on our approach play an important role in our context:

C1 It must scale to large systems with about 800 tables and tens of thousands of users.
C2 It must not require a global coordination because feature packs are developed by customers and third party vendors independently.
C3 It should be easy to use for developers at all participating organizations. This implies as far as possible the use of established technology like O/R-mapping in the application and standard SQL at the database level.
C4 It must be possible to modify attributes originating from multiple different feature packs in one transaction without creating lock conflicts.

4 Solution Approach

Based on the requirements and constraints, which we defined in the previous section, we will now discuss the approach, that we developed to address these problems. While our focus is on the general approach, we will also illustrate its specific application in the case study. We will first discuss how the customizability impacts the object-model, then we will describe how this leads to the management of variability on the data-model and database level. Finally, we will discuss the life-cycle of a feature pack in an installation and the corresponding database adaptation.

4.1 Object Model Design for Feature Packs

In this section, we discuss our approach to manage the object-model as a basis for supporting the development of features by multiple, independent organizations without full synchronization. Our approach to object model management indirectly also provides the basis for the customization of the database schema based on O/R-mapping as we will discuss in Section 4.2.

We use object oriented design (OOD) patterns to implement features. This is one of the common approaches to implement software product lines according to [6]. Unlike conditional compilation and aspect oriented programming, OOD enables the definition of stable interfaces. This is important because the final product is assembled by the customer based on fragments provided by independent organizations.

Delegation based design patterns allow multiple features to enrich model-objects with additional attributes without having to be aware of other enrichments. For example the *Project Output Measurement* and the *Publication Quality Assessment* both enrich the Publication-entity independently of each other. Users may still access both features together to edit a Publication-object because delegation is used instead of inheritance.

Figure 3 shows a class diagram of these enrichments. The Publication-entity is enriched independently by the *Project Output Measurement* on the left side and the *Publication Quality Assessment* on the right side.

As a consequence of this implementation approach, there is no need at the database layer to support adding of columns to tables from another feature pack. The additional classes are mapped to new tables instead.

4.2 Managing the Data-Model

Data modeling aims at developing a consistent database scheme that fulfills the domain requirements without introducing redundancies. Ideally, this is achieved by enforcing a global data-model. In our case, this is not possible, as we need to support distributed development of integratable data-models without global coordination, as required by the nature of software ecosystems. As platform providers HIS often never learns about certain extensions of the data-model created by customers or third-party suppliers.

Locally Coordinated Development of Data-Models: (Partial) database models are created by the core development organization, several third party vendors and customers. Although they are not coordinated globally except for the assignment of namespaces, they are typically coordinated on a per-organization-basis. For example, HIS consolidates all data for all its features in a single database scheme. While this internal coordination is not necessary for our approach, it is useful in practice, as it reduces redundancies, ensures naming conventions and prevents conflicting designs.

Splitting the Model: If organization-wide data-models are used, they are split into feature-related parts. Database tables are linked to their model classes by object/relation mapping definitions. As the model classes are part of exactly one feature pack, it is defined which database table belongs to which feature pack. Thus the variability dependency management implies relations among the database tables and supports their consistent combination.

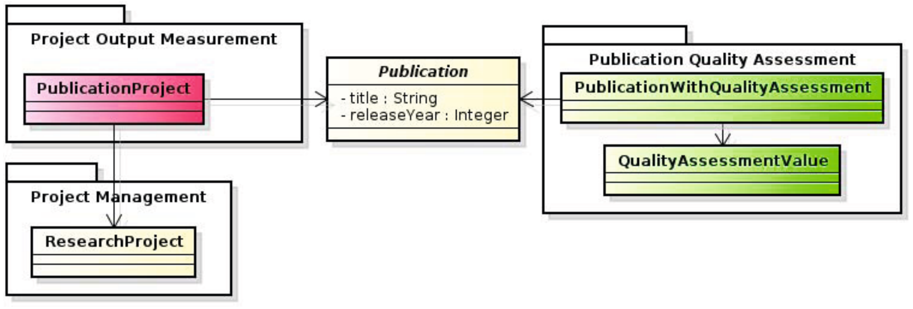


Fig. 3. Integration of the object model with entities provided by two feature packs

This mapping strategy requires that data is not stored in large monolithic tables, but tables are split up. Corresponding model classes, which only contain the relevant fields, must be introduced. (For example instead of having one huge table with many columns for all student data, several small tables such as *person*, *address*, *student_information*, *library_id*, *fee_credit* are used). Features, that must access this information, need to mark the corresponding feature packs (and hence database tables) as required.

We usually attach foreign-key constraints to the tables they are defined on. But in some cases, a feature pack may introduce a constraint on a table from another feature pack. In such cases the definitions must be assigned to the feature pack explicitly. Views can also be attached to the first tables, in the same way, and they too may be assigned to a different feature pack explicitly. While views are not commonly used in applications, which are based on object/relational mapping, some customers like to use them as an easy means to extract information from the database without writing program code.

Example of the Global View on a Partially Coordinated Model: Figure 4 illustrates our approach based on the example from Section 2.2. The development organization maintains an organization-wide database model, which is shown on the left side of the figure.

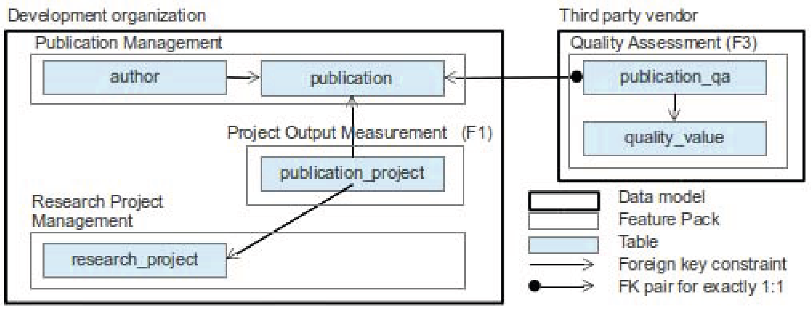


Fig. 4. Distributed database design with coordination on a per organization level

The features *Publication Management*, *Project Output Measurement (F1)* and *Research Project Management* are managed by the development organization. The corresponding data is stored in the tables *author*, *publication*, *publication_project* and *research_project* and maintained by the development organization. The foreign-key relation between *author* and *publication* belongs to the feature *Publication Management*. The feature *F1* contains the foreign-keys from *publication_project* to *publication* and *research_project*.

A third party vendor develops the feature *Publication Quality Assessment (F3)*, which contains the two tables *publication_qa* and *quality_value*. This is shown to the right of Figure 4. *F3* also provides a pair of foreign-key constraints between *publication_qa* and *publication* in both directions to ensure an one-to-one relationship.

Data Model Format: The data-models and partial data-models are described using a subset of Structured Query Language (SQL). Thus common modeling tools can be used as SQL is widely supported.

The subset consists of `create` statements for tables, views and indexes. For constraints the syntax in SQL starts with `alter table add constraint`. All other `alter table` commands are explicitly disallowed. Triggers and stored procedures are not used in our context, but may be defined using `create` statements.

During update or installation of a feature pack, the SQL-statements are not executed, but analyzed as a description of the desired reference model. The necessary actions to alter the structure of the existing database to the desired model are derived from a comparison of both definitions, as we will discuss in Section 4.4. This process also takes vendor-specific incompatibilities of SQL dialects into account by creating appropriate SQL-statements internally.

4.3 Feature Lifecycle Support

Over time feature packs may be added, removed or updated in a given installation, which may result in an adaptation of the data-model. Figure 5 shows the states and transitions of the feature life cycle in an installation. We will discuss the transitions in the context of the three processes of adding, removing and updating a feature pack.

Adding a New Feature Pack: Initially the new feature pack does not exist in the installation. The feature pack is *uploaded* by an administrator. This step makes the feature implementation and data-model definitions available to the system. During the *install* step, the partial data-model provided by the feature pack is integrated into the database. We will discuss this process in detail in Section 4.4. After the feature pack was installed, it will be *activated*, which allows people to use it.

Some features require data, that can only be provided by domain experts manually. Even when these features are in state active, they can only be used in a limited fashion until the necessary data is provided. For example the quality assessment of publications is not visible to the general public until the domain experts have assessed the publications. This half activated state is managed by the features themselves, similar to common domain-inspired situations in which some information is hidden from certain users.

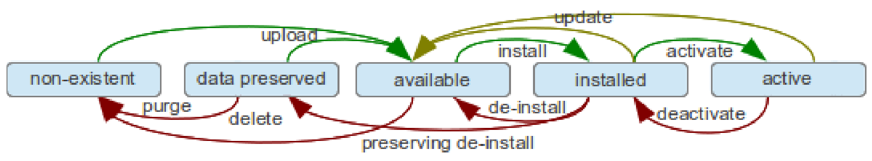


Fig. 5. Life-cycle of a feature pack

Removing a Feature Pack: Feature packs in any state may be removed from an installation, which will result in the following transitions.

A feature pack, which is in state active, is *deactivated*, resulting in the installed state. This transition does not influence the database and is therefore suitable for temporarily hiding a feature. An installed feature pack must be *de-installed* to adapt the database to the structure without the data-model part of the de-installed feature pack. This results for the feature pack being in state available. An available feature pack may be *deleted* without any further impact on the database, because the data and the structure has already been removed on *de-installing* the feature pack.

Mass deletion of data is a very dangerous action as it cannot be undone easily. Thus, it is desirable to be able to perform a *preserving de-install*, which keeps the data of a feature. This preserved data, however, will decay over time without constraints and without the program logic to maintain it. But the ability to undo a de-installation by uploading the feature pack again is highly desirable, because the alternative of merging the current database with a version restored from a backup, is a highly complex task.

After it is ensured, that the preserved data will not be used anymore, it may be *purged*. Or the feature pack may be *uploaded* to make it available again.

Updating a Feature Pack: An *update* of an available, installed or active feature pack is done by uploading a newer version of the feature pack, which replaces the old one and makes the newer version available in the system. The transitions from available through installed to active are exactly the same as in the process of adding a new feature pack. The main difference is that the old feature pack is not removed prior to integrating the new one. But the combined effect is determined and only those changes are applied to the database as part of the install-transition. The details of the activities leading to the adaptation of the data-model are described in the following section.

4.4 Customizing the Database

The integration of the partial database models provided by the available features into the actual database is a process that consists of four phases: *Combine partial data-models*, *initial structure adjustment*, *data modification* and *final structure adjustment*. We will discuss these phases below, describing initial and final structure adjustment together. The database adjustment process is started by an administrator. The execution of the four phases is a fully automated process.

Combining the Partial Data-Models: In the first phase of the database adaptation, the desired data-model for the complete database is derived based on a combination of the partial data-models of feature packs in state *data preserved* and later. This combination is given as the set union of all tables, columns, constraints, and views defined by any of the partial data-models. A naming conflict, which can be the result of an element being renamed or independent introduction of elements with the same name, is prevented by the coordination within an organization and namespaces across organizational borders. As each column and table is defined by exactly one feature pack as discussed in Section 4.2, there is no redundancy that needs to be addressed on this level. As far as necessary, this is handled on the variability model level.

The initial version of the combined data-model is not necessarily consistent. It may contain foreign-key constraints for which the source or target table is not part of the combined model or views using non-existing tables. These dangling constraints and views are the result of unfulfilled optional dependencies on the feature pack level. This is illustrated by our example, where we implemented the Research Output Measurement as a new feature pack, which depends on the feature packs Publication Management and Research Project Management. Alternatively the functionality could be implemented as part of Research Project Management with an optional dependency on Publication Management. Here, the Research Project Management would introduce a foreign-key constraint on Publication Management. However, this constraint would be eliminated in installations where Publication Management would not be present.

For feature packs, which are in the state *data preserved*, only the information regarding required tables and columns of the database is included. Information on constraints are ignored, as the implementation, that would be observed is not part of the running system. For example, there might be a constraint enforcing that every publication is related to a project, but if Research Project Management would be in the state *data preserved*, this could no longer be managed by the system, thus the constraint may not be enforced any longer.

Structure Adjustment: The structure (tables, columns, constraints, indexes and views) of the actual database is adjusted to meet the desired combined data-model. The structure of existing data may change, for example a column may be moved into a new table. The data modification phase needs to have access to both the old data and the new target structure. This requires the extension of the existing database structure with new elements during an initial structure adjustment phase before the data modification phase. The final structure adjustment narrows the database structure down to fit the desired data-model.

The adjustment is based on the results of a comparison between the actual database structure and the desired model for all element types except views. Table 2 shows all possible situations.

An element may be absent in the actual database structure but present in the desired one, which means an element is new. The opposite situation occurs when an element exists in the database, but not in the desired model. Finally an element may be present in both models. In this case both definitions of the element have to be compared in detail. For tables this means that the order of columns may need to be adjusted. For columns, changes of datatypes must be applied. Indexes and constraints are identified by their complete definition. So a modification of such an element type results in the treatment as a deletion and a creation. This is useful because common database systems do not allow the modification of constraints and indexes without their recreation.

For simplicity all views are dropped in the beginning and recreated in the end, because views prevent modifications of the underlying tables in all common database systems. Since views do not contain any data directly and their creation and removal is very fast, this is a simple and effective approach.

In the **initial structure adjustment** phase additional tables, columns and non-unique indexes increase the possibilities of the data-model and are therefore created as shown in Table 2 a). Constraints and unique indexes, however, limit the possibilities of the data-model, therefore outdated ones are deleted as shown in Table 2 b). The order of columns will be fixed. If the datatype of a column does not match and the target type is wider, it will be adjusted.

In the **final structure adjustment** phase, deleted tables, columns and non-unique indexes are removed. Remaining datatype changes in columns are performed. All views, new constraints, and new unique indexes are created.

The creation of some new constraints may not be possible because existing data may not comply. In most cases default values will be set in the data modification phase before the constraints are defined. But sometimes expert knowledge is required to determine adequate data. In this case the constraints cannot be enforced during the installation of the feature pack, but must be added at a later time. The affected feature packs must provide functionality for domain experts to enter the necessary information, while disabling functionality, which would require this data. When all data conforms to the constraint, the database adjustment process may be restarted and will add the missing constraints.

Data Modification: In the third phase of the database customization, the data is converted to fit the new structure and requirements. The data modification is defined as a series of actions, which rely on a set of standardized, common patterns, which were identified based on experience at HIS over roughly a decade. They are listed in Table 3.

As special cases may arise, that cannot be addressed by the common standard patterns, the list also includes two actions which are rather generic. The first allows arbitrary SQL-statements, while the second even allows arbitrary program code. However, using the patterns for common cases is preferable, as it improves maintainability. Furthermore it abstracts from incompatible SQL-dialects used by different vendors, for example for multi table update statements.

The actions may have preconditions like the existence of a specific column or table. For example renaming of a column includes a *Copy column* step, that is executed, if both the old column and the new column exist during the data modification phase. The

Table 2. Required actions based on the results of the database comparison

a) Tables, columns, indexes

New / Old	Absent	Present
Absent	-	Create in phase 2
Present	Drop in phase 4	*+

b) Constraints, unique indexes

New / Old	Absent	Present
Absent	-	Create in phase 4
Present	Drop in phase 2	+

*) Order of datatypes and columns is adjusted
+) Indexes and constraints are identified by their definition

Table 3. List of common data modification patterns

Name	Description
Load default data	Load default data into a new table or column.
Add default data	Add rows of default data into an existing table, unless those rows already exist.
Copy table	Copy all rows from an old table into a new one.
Copy column	Copy all cells from an old column into a new one.
Move column	Fill a new table based on the cells of the old columns and the corresponding row id as foreign-key.
Introduce key table	Fill a new table based on unique values from an old column and write the new row id into a new foreign-key column of the corresponding row in the source table.
Disambiguate sort order	Disambiguate integer values in a column across rows, within groups formed by other named columns.
SQL statements	Executes specifically written database statements.
Program code	Executes specifically written program logic.

new column was created by the initial structure adjustment and the old column will be deleted by the final structure adjustment.

5 Evaluation

In this section we will discuss the evaluation of our approach. We will first describe the different evaluation activities, we performed, and then discuss our findings in relation to the research questions from Section 3.

5.1 Evaluation Activities

We evaluated our approach in several ways, as we will describe in this section. Multiple evaluation activities may relate to the same contribution of our approach, as we will describe below. We start our description of the activities with the most recent effort.

E1: Over the last half year, we (the first two authors of this paper) developed a prototype to evaluate the possibility of partitioning the data model into fine-grained, feature-specific data models by decomposing the corresponding object structure. This prototype was implemented in the context of a current feature development at HIS: the Research Management domain. It resulted in the decomposition of the features as described in Section 2.2 while investing an effort of about two weeks of work. The evaluation was used on the one hand to understand better any (technical) limits on choosing fine-grained features and on the other hand as basis for discussions on the approach with developers, the software architects and product managers.

E2: A fellow developer created a feature to manage test-case descriptions with an effort of about half a year distributed over the course of a year. The goal of the implementation was to show that combining data-models works in practice for large partial data-models without coordination. The feature has been in production internally for over a year, and was updated several times. With fifteen database tables, it is significantly larger than the prototype features.

E3: We gathered the experience, HIS gained in evolution and ecosystems with information systems in production over the last ten years, by talking to customers, support, product managers and fellow developers. According to [13] gathering expert opinion is the state of the art approach to estimate maintainability in database based applications. We looked especially at the database update process and the combination of multiple, partial data-models. Further, we wanted to learn about any existing issues.

E4: We analyzed HISinOne to estimate the performance impact of the decomposition approach. HISinOne is a six year old University Management System, which is currently in production by 53 customers. We collected and calculated metrics on the database structure as shown in Table 1. We measured the performance of the database adjustment tool and looked for indications of a possible performance impact by having many small tables.

5.2 Separating Data-Models

The decomposition of organization-wide data-models into small, feature-specific models and the management on this level (RQ1) was evaluated by E1, E2 and E4.

Observation and Discussion: A possible problem with using delegation to extend model objects from other feature packs (cf. Section 4.1) is the increased complexity. E2 showed that this is not significant while E1 confirms this for small feature packs.

We found no negative impact on performance by using many small tables based on E4. The core objects of HISinOne (e.g., applications, students, exams) have 7 to 15 columns with an average of 7.8 columns per table. At a university with about 40 000 students, the database contains an overall sum of 17 284 995 records for the student management domain. The largest ten tables cover a range of about 180 000 to about 6 million records[1]. Typical installations in such a context use 2 to 20 application servers to handle the load during peak times, while only one database server is sufficient. Since the existing modeling is already based on small tables without a performance bottleneck, we expect no problems from using small tables motivated by feature packs.

Limits: The organization-wide coordination of data-models may be seen as a limit. In discussions E1 and E2 with developers, they welcomed a coordination across features and domains as it simplifies maintenance and prevents redundancies. But, our approach does not depend on organization-wide harmonization as feature-specific data-models would be sufficient.

Threats to Validity: E1 was implemented by the first two authors of this paper, but it was extensively discussed with other developers and E2 was created by a different developer, reducing threats to external validity. E2 and E4 concern systems that have been in production for more than a year, respectively for many years, thus we don't

[1] Data measured in a anonymized copy of a production database from a university with about 40 000 students.

see any threats. In particular, based on the ecosystem experience we do not assume significant issues in terms of external validity.

5.3 Combining Partial Data-Models

We evaluated the process of combining of partial data-models in production environments at our customers (E3) and looked at the performance impact (E4).

Observation and Discussion: Combining partial data-models from multiple sources in concrete installations, as discussed in Section 4.4, works fine according to our experience with E3. It was implemented three years ago and is used by customers and third party vendors in production since then.

The performance impact of the composition algorithm is negligible in practice. It takes less than 50 milliseconds to combine the data-models in the installation we looked at. The algorithm uses a simple set union for all elements, and then iterates over all views and foreign-key constraints doing a hash-set-lookup for elements, they depend on, to clean up the model. As this scales with $O(nlog(n))$, we conclude that the performance impact even for a large amount of feature packs is still acceptable.

Limits: In order to prevent conflicts across organizations, our approach requires the assignment of namespaces for tables, columns, indexes and constraints. The management of dependencies among partial data models needs to be handled with another technique, such as admin guide lines or feature pack dependencies.

Threats to Validity: We did not actually test a large number of feature packs, but so far our experience is very positive that the performance only depends on the number of elements and is negligible in practice. This is supported by our analysis of the complexity of the composition algorithm.

5.4 Database Adjustment

We evaluated whether the database adjustment fulfills the requirements of installing and updating feature packs in installations (RQ3).

Observation and Discussion: The database adjustment process from Section 4.4 is in use for ten years with only minor improvements since then (E3). The data modification patterns listed in Table 3 evolved over time, but are stable since four years. The tool is used about 200 times per year in production systems by 90 customers for two different, large-scale information systems. It is also used several hundred times in development and for prototype testing by customers. It is quality tested on Informix and PostgreSQL, but implementations for other database systems exist as well.

Compared with the previous approach the number of failed updates, especially for prototypes and hot-fixes, was reduced significantly by the new approach, according to customer support (E3).

Lesson Learned: While we learned some lessons since the database adjustment tool was first used in production in 2002, they have not required fundamental changes to the process: Support for views was added about five years ago, due to customer interest.

Deletion of tables and columns is recently very rare, but occurred more often during the initial development of HISinOne about 4 years ago.

Limits: Triggers and stored procedures are not used by HIS. If the same database structures are changed repeatably and customers skip versions, data modification rules may need to be modified. We have not automated this, yet, because this situation is not relevant in practice according to E3.

Threats to Validity: We do not assume significant threats as this approach is used extensively in production.

6 Related Work

Our approach allows the evolution of the database in an information system ecosystem, which is little covered by research yet.

In [7] the author discusses a set of migration patterns, which is suitable for the migration from an older system to a newer system and is similar to the data-modification-patterns explained in Section 4.4, but without tailoring the database structure. An approach for refactoring existing database structures based on defect patterns is discussed in [8]. This approach especially aims at improving the database structure by refactoring, but the existing semantics of the model is kept.

A customizable database schema evolution approach for object-oriented databases is discussed in [12]. It offers support for both schema evolution and adjustment of existing data, but the tailoring of the schemes is not covered.

Several approaches for tailoring of database schema exist. In [14, 17] the use of database views for tailoring is proposed, yet no support for evolution of the scheme is offered. The use of a global database model, which is pruned to the actual needs, is discussed in [1]. Another approach based on a global database model, whose components like columns are overloaded for customizing it, is discussed in [18], but evolution is not covered by this approach. In [10] Mahnke proposes the use of component-based database schema development, but does not cover evolution of the schema or of the data. Dyreson and Florez discuss a specialized approach for adding cross-cutting concerns to data-models with aspects [5]. Yet, it does not cover evolution and is focused on those cross-cutting concerns. Sabetzadeh et al. describe an approach for the composition of models from parts with the goal to enable global consistency checking, which does only cover models and does not consider evolution [15]. Finally Schaeler et al. split a global database model into parts with the help of SQL analysis and a manual clean up. The approach does not consider the evolution of the models and the corresponding data.

All related work has in common that the approaches do not origin from an ecosystem context. The ecosystem perspective seems to be mostly unique to our approach.

7 Conclusion and Future Work

In this paper we described an approach to database management and evolution, which supports customization and (partial) evolution of the database schema. This approach relies on core functionality that has been production use for many years, with nearly a hundred different customers. However, more recently, we extended it with capabilities that support the more fine-grained evolution and customization. Moreover, as opposed to other, related approaches, our approach addresses software ecosystems and works well in this context. We evaluated our approach using several independent evaluation approaches and discussed the conclusions we can draw from them.

In the future, we are going to further improve feature pack support on the application level.

In particular, we envision the integration of an explicit dependency management support, which is so far not yet supported. We plan to achieve this by integrating our approach with a corresponding dependency management approach that we described earlier [4]. It relies on the idea of a distributed variability model, as we discussed in [16].

References

1. 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)
2. Bosch, J.: From software product lines to software ecosystems. In: 13th International Software Product Line Conference, pp. 111–119 (2009)
3. Brummermann, H., Keunecke, M., Schmid, K.: Variability issues in the evolution of information system ecosystems. In: 5th Workshop on Variability Modeling of Software-Intensive Systems, pp. 159–164 (2011)
4. Brummermann, H., Keunecke, M., Schmid, K.: Formalizing distributed evolution of variability in information system ecosystems. In: 6th International Workshop on Variability Modeling of Software-Intensive Systems, pp. 11–19 (2012)
5. Dyreson, C., Florez, O.: Data aspects in a relational database. In: 19th ACM International Conference on Information and Knowledge Management, pp. 1373–1376 (2010)
6. Gacek, C., Anastasopoulos, M.: Implementing product line variabilities. In: Symposium on Software Reusability (SSR 2001), pp. 109–117 (2001)
7. Haller, K.: Towards the industrialization of data migration: Concepts and patterns for standard software implementation projects. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 63–78. Springer, Heidelberg (2009)
8. Lemaitre, J., Hainaut, J.-L.: Transformation-based framework for the evaluation and improvement of database schemas. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 317–331. Springer, Heidelberg (2010)
9. Maes, P.: Intelligent software. In: 2nd International Conference on Intelligent User Interfaces, pp. 41–43 (1997)
10. Mahnke, W.: Towards a modular, object-relational schema design. In: 9th Doctoral Consortium at CAiSE (2002)
11. Pleuss, A., Hauptmann, B., Keunecke, M., Botterweck, G.: A case study on variability in user interfaces. In: 16th International Software Product Line Conference, pp. 6–10 (2012)
12. Rashid, A.: A framework for customizable schema evolution in object-oriented databases. In: 7th International Database Engineering and Applications Symposium, pp. 342–346 (2003)
13. Riaz, M., Mendes, E., Tempero, E.: Towards predicting maintainability for relational database-driven software applications: Extended evidence from software practitioners. International Journal of Software Engineering and Its Applications 5(2) (2011)
14. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requirements Engineering 11, 174–193 (2006)

15. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: International Requirements Engineering Conference, pp. 221–230 (2007)
16. Schmid, K.: Variability modeling for distributed development: A comparison with established practice. In: 14th International Conference on Software Product Line Engineering, pp. 155–165 (2010)
17. Spaccapietra, S., Parent, C.: View integration: A step forward in solving structural conflicts. IEEE Transactions on Knowledge and Data Engineering 6(2), 258–274 (1994)
18. Ye, P., Peng, X., Xue, Y., Jarzabek, S.: A case study of variation mechanism in an industrial product line. In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 126–136. Springer, Heidelberg (2009)