Database Evolution for Software Product Lines

Kai Herrmann[1], Jan Reimann[2], Hannes Voigt[1], Birgit Demuth[2], Stefan Fromm[3], Robert Stelzmann[4] and Wolfgang Lehner[1]

[1]*Database Technology Group, Technische Universität Dresden, Germany*

[2]*Software Technology Group, Technische Universität Dresden, Germany*

[3]*Dresden-Informatik GmbH, Dresden, Germany*

[4]*iSAX GmbH & Co. KG, Dresden, Germany*

Keywords: Database, Software Product Lines, Evolution.

Abstract: Software product lines (SPLs) allow creating a multitude of individual but similar products based on one common software model. Software components can be developed independently and new products can be generated easily. Inevitably, software evolves, a new version has to be deployed, and the data already existing in the database has to be transformed accordingly. As independently developed components are compiled into an individual SPL product, the local evolution script of every involved component has to be weaved into a single global database evolution script for the product. In this paper, we report on the database evolution toolkit DAVE in the context of an industry project. DAVE solves the weaving problem and provides a feasible solution for database evolution in SPLs.

1 INTRODUCTION

With the directives 2008/43/EG and 2012/4/EU, the European Commission made the tracking of explosives mandatory in the European Union (EU) from 5th April 2015 on. All explosives for civil use, including detonators, primers, boosters, cords, etc., have to be tracked during their whole life cycle in the EU. This has to be applied from the manufacturing location or import into the EU towards the end user. For instance, a company producing black powder has to attach a unique identifier to every unit it produces. A freight shipping company transfers such units of black powder to distributors. Another manufacturer buys a unit of black powder to make primers, each getting a new identifier. The life cycle of the primer and the black powder continues to include further dealers and carriers until they are eventually put in use in e.g. a small stone quarry. All participants in the life cycle of explosives are required to store the tracking information of each item as it passes their domain. The tracking information includes the identifier of each item together with the time stamp and partner of the incoming and outgoing events. In the research-oriented industry project euroTRACKex (http://www.tt-e.eu) (eTe), we develop a demonstrator of a tracking software for explosives that will help companies to fulfill their obligations.

Independent of the application field, life cycle tracking confronts the involved parties with liabilities, which can only be handled feasibly with IT support. All participants need basically the same software to track specific goods. However, their detailed requirements vary considerably. Participants (small business to large enterprise, producer to dealer to carrier to consumer) significantly differ regarding their financial constraints, followed processes, implemented IT landscapes, and national legislations.

With such a diverse customer base, it is economically infeasible to try designing and implementing one software product that satisfies the requirements of all potential customers. Likewise, it is uncompetitive to implement a specialized solution for each customer. Software product line engineering is a long studied but rarely implemented technique to gain the necessary flexibility in software development for handling a very diverse customer base. An SPL is defined by a common software model, which decouples the development from the deployment. That allows distributing the development of the software components among multiple parties. Ultimately, concrete software products can be generated from the SPL according to an individual configuration for each customer. Software technology research provides tooling and meth-

Herrmann K., Reimann J., Voigt H., Demuth B., Fromm S., Stelzmann R. and Lehner W..
Database Evolution for Software Product Lines.
DOI: 10.5220/0005484101250133
In *Proceedings of 4th International Conference on Data Management Technologies and Applications* (DATA-2015), pages 125-133
ISBN: 978-989-758-103-8

ods to realize SPLs. Hence, SPLs are the technologies of choice in the eTe project.

The problem of economic feasibility in the development process aggravates as software evolves inevitably (Lehman, 1980). SPL components are enhanced, fixed, and updated. While evolution is an important aspect of SPL research, the database layer is typically not considered and vice versa. However, in evolving SPLs the evolution of the database layer becomes a central problem (Terwilliger et al., 2012; Roddick, 1995). With every deployed new product version, the database schema may change and the existing data has to be transformed accordingly. The problem aggravates if the SPL components are implemented by independent parties, each having a local view. The database has to evolve globally, in one step, as the product evolves. The decoupling of development and deployment in SPLs poses a new challenge to database evolution: When a customer's product evolves, many locally specified evolution steps have to be weaved into a single global database evolution script. This is called the *weaving problem*.

The actual extent of the weaving problem significantly depends on the chosen database system. Relational systems ensure a strictly structured schema, hence, the weaving of evolving components becomes very complex. On the contrary, NoSQL systems keep the data in a more flexible structure, which simplifies the weaving problem by design. However, NoSQL stores are still subject to the weaving problem. We use a relational database, since many of its well established features are indispensable for our project and worth the effort to solve the weaving problem for relational databases. Another requirement posed by our industry partners is a lean architecture at runtime. The products generated by the SPL are normal database applications without any additional layers.

In this paper, we report on the eTe database evolution toolkit DAVE (DAtabase eVolution for tracking of Explosives). DAVE solves the logical level of the weaving problem, hence, performance optimization and evaluations are out of scope. We introduce SPLs as known in the software technology community in Section 2. In Section 3, we describe the evolution of SPLs and the resulting weaving problem. We present DAVE in Section 4. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 SPL ENGINEERING

As explained in the previous section, the initial situation of the eTe project was, that the software vendor has many different customers each requiring similar software systems. All of them contain the same functional core but may differ in quantity or presence of other functional components. All customers can decide independently which components to select for their business. Such related software systems are called a *software family*. Obviously, this scenario contains a dimension of variability which is necessary to be controlled and managed. It has to be avoided that, for a new customer, an existing product is replicated and then customized because of the single source principle. It is hardly possible to consistently fix potential bugs in each and every customer product which may differ only slightly. This unmanaged redundancy results in huge maintenance effort (Murer et al., 2010). As a consequence, the techniques and methodologies of SPL engineering (Pohl et al., 2005) are applied in software development.

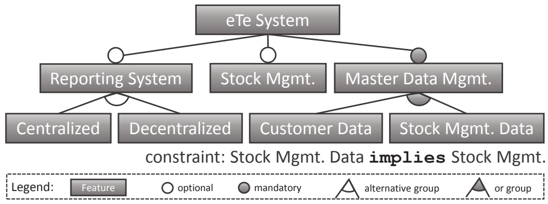


Figure 1: Subset of the eTe feature tree.

Software product line engineering defines a family of closely related software systems consisting of common and variable functionality in order to manage variability. It aims at separating configuration knowledge, regarding what functionalities belong to a concrete product, from the actual realization of that product. Thus, one main benefit of SPL engineering is that configuration knowledge is captured on a conceptual non-technical level, and hence can be accessed by non-programmers easily. Therefore, *problem space* and *solution space* are distinguished. The former consists of the conceptual configuration knowledge and the latter contains the artifacts realizing desired functions (Czarnecki and Eisenecker, 2000).

The variability in the problem space is commonly described with *feature models* (Kang et al., 1990) wherein *features* are arranged in a *feature tree* (Chen et al., 2005). Selecting one feature in the tree automatically selects its parent feature. To determine if a feature is *mandatory* or *optional* it can be marked as such. Furthermore, features can be combined into *or groups* or *alternative groups*. The former allows selecting at least one child feature, whereas exactly one feature must be selected (in terms of an `xor` selection among the child features) from the latter. Beyond that, *cross-tree constraints* can be specified over the features to express relations not being able to be reflected in the tree structure (Batory, 2005).

In Figure 1, a small subset of the eTe feature

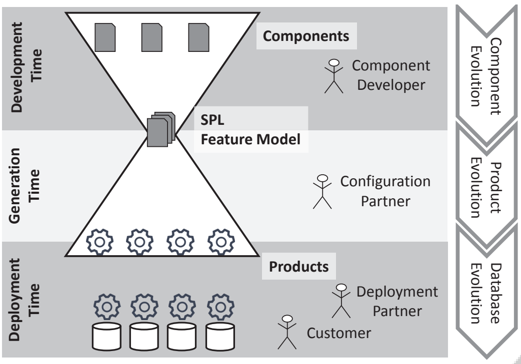


Figure 2: SPL engineering process.

tree is illustrated. It contains the root feature `eTe System` having the mandatory child feature `Master Data Mgmt`. Its children (`Customer Data` and `Stock Mgmt.Data`) reside in an *or group*, thus at least one of them must be selected. The two features `Reporting System` and `Stock Mgmt` are optional. If the former is selected, then either a `centralized` or a `decentralized` reporting system needs to be selected, since both features are contained within an *alternative group*. Furthermore, the cross-tree constraint ensures that if the `Stock Mgmt.Data` feature is selected, the `Stock Mgmt` feature must be selected, too. A feature tree represents all possible configurations whereas a *configuration* is a valid subset of all features satisfying all constraints and selection rules. Thus, a feature model is a compact and concise notation of a large number of possible configurations.

Transforming a conceptual configuration into an executable software system poses two prerequisites. First, a *mapping* from problem space to solution space must be specified to define the semantics of the particular features. Within the eTe project, we decided to establish a 1:1 mapping of feature (problem space) to software component (solution space) for the sake of simplicity. Second, a *variability realization mechanism* is needed producing the final product w.r.t the configuration. The final product is called a *variant*.

3 EVOLUTION OF SPLs

SPLs allow generating new products whenever the customer's requirements change or new versions of chosen features are published. This flexibility hits the wall at the database layer. Typically, customers want to keep their data when updating their products. Inevitably, evolution in SPLs includes the evolution of databases, which is still a major headache in practice. SPLs involve three major program life cycle phases: *development time*, *generation time*, and *deployment time*, as illustrated in Figure 2. All three phases are decoupled regarding when they happen (time) and in whose domain they are performed (space).

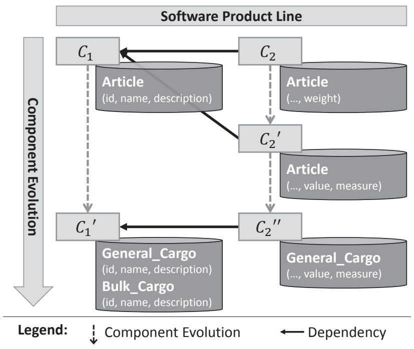
At development time, a *component developer* implements a component that realizes a specific feature in the SPL. Components are purely additive and do not alter the other components, but they may extend others. If a component requires persistence, the developer defines the corresponding data model for the component. To make a component available, the developer submits the code and a formal component description (name and its dependencies to other components) to a central component repository.

At generation time, a *configuration partner* selects a specific set of features required by a customer. A configurator tool provides a convenient UI for this task. Once a variant of the SPL is configured, the configurator compiles the product desired by the individual customer by resolving the selected features to software components w.r.t. the mapping between problem and solution space. The database schema for the product results from the union of the database schemas of the selected components.

At deployment time, a *deployment partner* deploys an individual product on its runtime platform and makes it available to the customer. Since product deployment is decoupled from component development, many individual products can be deployed easily for very different customers. During product deployment, the database is set up and tables are created according to the product's database schema.

Evolution can occur in all three phases. At development time, developers improve, update, refactor, and debug their components including the underlying data model. We call this *component evolution*. At generation time, customers request reconfiguration of their products, because they want to add/remove components or update to a new component version, resulting in *product evolution*. At deployment time, we have to consistently evolve an existing database, including schema and data, according to the new product version. This is *database evolution*. Database evolution is necessary if the data model of a product changes in component or product evolution and these changes are actually rolled out to the customer.

Consider the small example in Figure 3. It shows the component evolution of the components C_1 and C_2 with their respective data models. The data model of C_1 consists of a table `Article` with three columns. C_2 builds on that data model and adds the column `weight` to `Article`. Say a customer runs a product with configuration $\{C_1\}$ and wants to change to configuration $\{C_1, C_2\}$. In case of this product evolution, the database has to evolve, too. After adding the col-



$\{C_1\} \rightarrow \{C_1, C_2\}$
$\{C_1\} \rightarrow \{C_1, C_2'\}$
$\{C_1\} \rightarrow \{C_1'\}$
$\{C_1\} \rightarrow \{C_1', C_2''\}$
$\{C_1, C_2\} \rightarrow \{C_1\}$
$\{C_1, C_2\} \rightarrow \{C_1'\}$
$\{C_1, C_2\} \rightarrow \{C_1, C_2'\}$
$\{C_1, C_2\} \rightarrow \{C_1', C_2''\}$
$\{C_1, C_2'\} \rightarrow \{C_1\}$
$\{C_1, C_2'\} \rightarrow \{C_1'\}$
$\{C_1, C_2'\} \rightarrow \{C_1', C_2''\}$
$\{C_1'\} \rightarrow \{C_1', C_2''\}$
$\{C_1', C_2''\} \rightarrow \{C_1'\}$

Figure 3: Example for evolution in SPLs.

umn `weight`, the evolution has to add `weight` values for all existing articles, e.g., by inferring `weight` values from the product description.

Later on, a new version of C_2, called C_2', is released. Instead of `weight`, C_2' uses the two columns `value` and `measure` to represent the weight of an article. This component evolution does not cause a database evolution as long as no product is evolved to include C_2'. If a product evolves to include C_2', the new columns `value` and `measure` of already existing articles must be populated, e.g., splitting existing `weight` values (e.g. 5 kg, 500 g) into `value` and `measure`.

In another component evolution, C_1 is updated to C_1'. For C_1', the component developer decides to change the data model. Because general cargo and bulk cargo are often handled separately, the developer horizontally partitions the `Article` table into the tables `General_Cargo` and `Bulk_Cargo`. C_2' is not compatible with this new version C_1'. The developer of C_2' reacts and updates the component to C_2''. In C_2'', the additional columns only extend `General_Cargo` since bulk cargo does not have any fixed weight.

As shown in the example, the data transformation for an evolution step, e.g., inferring new values or splitting existing values, depends on the application logic that uses the data. In the context of SPLs, only the component developer knows the specifics of the application logic and is able to specify the data transformation necessary for the component's evolution.

The example in Figure 3 already involves five possible configurations of a product and thirteen possible product evolutions, in total. The large – in the extreme case exponential – number of possible configurations and product evolutions in SPLs is intended by design. The combinatorial explosion is the true power of SPLs and allows providing highly individual products to customers. Nevertheless, it is absolutely infeasible for a component developer to consider all product evolutions a component may be involved in. In SPLs, database evolution can be specified only locally at development time with a scope limited to the new component, its predecessor, and all components the new component depends on.

At deployment time, when database evolution actually happens, the locally defined evolution scripts form one global evolution script. The global evolution script carries out the database evolution for one specific product evolution. The key challenge to implement database evolution for SPLs is to weave all relevant local evolution scripts to one global evolution script. We call this the weaving problem.

Consider an evolution from $\{C_1, C_2'\}$ to $\{C_1', C_2''\}$ from Figure 3. The local script for the evolution $C_1 \rightarrow C_1'$ creates the two new tables `General_Cargo` and `Bulk_Cargo`, moves the data from `Article` to the new tables, and drops `Article`. The local script for the evolution $C_2' \rightarrow C_2''$ creates the columns `value` and `measure` in the `General_Cargo` table, moves data from the `Article` table to the new tables, and removes the columns `value` and `measure` from `Article` table. Obviously, these local evolutions have dependencies, which prohibit sequential execution.

The weaving problem has two aspects. On the logical level the global evolution script has to be correct. It must not result in a database different from the component developers' intent. For each component of the new product the resulting database has to provide the expected structures. On the physical level the global evolution script has to be efficient. It is efficient if it performs the necessary change to the database in the shortest possible time with a minimum amount of resources. DAVE solves the weaving problem on the logical level. The physical optimization of a global evolution script could not be addressed in the eTe project so far and is open for future work. Hence, there is no evaluation containing e.g. performance measures. We focus on the logical level and validate the feasibility within our industry project.

4 DATABASE EVOLUTION TOOLKIT DAVE

DAVE implements a demonstrator for the eTe SPL including a solution for the logical level of the weaving problem. In this section, we describe the developed process and its tool support. Well in line with SPL engineering, DAVE stores all required information for data management in an abstract format locally within each component. During the configuration and generation of a customized product, this information is simply collected from all participating components. Finally, DAVE generates one global database evolution script for each deployment, depending on the pre-

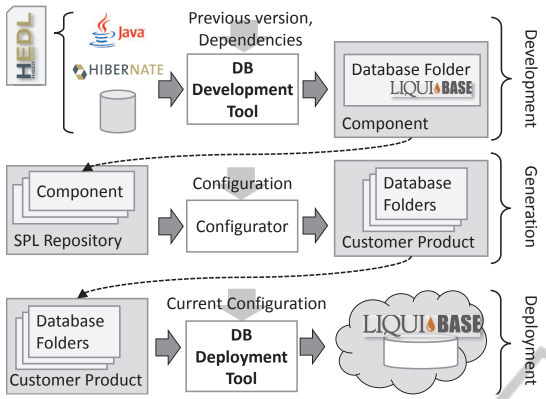


Figure 4: Data management process for SPL engineering.

vious and the new configuration.

The data management process of DAVE, as shown in Figure 4, is based on the general process description for SPL engineering in Figure 2. The developers use the domain-specific language (DSL) HEDL to describe the data model. When developing a component, we create a *database folder* containing the local database evolution steps, which are defined using *Liquibase* (http://www.liquibase.org). Liquibase simplifies the handling of database evolution, is independent of the concrete relational DBMS, and allows determining the difference between given schemas. The *DB Development Tool* creates the database folder based on the new component and its previous version, if existing. We discuss the DB Development Tool in Section 4.1. Generation time requires no additional database-related tool support. Basically, all database folders of the selected components are collected and included in the final customer product. At deployment time, the *DB Deployment Tool* weaves the collected information into one executable Liquibase script, as described in Section 4.2. The DB Deployment Tool is the heart of DAVE; this is where the weaving problem is solved.

4.1 DB Development Tool

The single components of an SPL are developed separately. It is unforeseeable, which other components will also be part of a deployed product. Nevertheless, these other components may use or extend the component's data model. The DB Development Tool of DAVE ensures, that all necessary information about a component is collected locally at development time to deploy any product globally.

Input. Developers specify a component's data model using Hibernate Entity Definition Language (DevBoost, 2013) (HEDL). HEDL is a DSL being able to generate the Java Persistence API (JPA) layer of an application, which is responsible for persisting and accessing data in a relational database. HEDL has a concise syntax for defining the persistence layer of a specific domain. A HEDL document is transformed to Java entity classes and data access object classes automatically. HEDL can be used for Hibernate or any other JPA implementation. For instance, the component C_1 from the example in Figure 3 is described as shown in Listing 1.

```
Article {
    String identifier;
    String name;
    String description;
}
```

Listing 1: HEDL model for C_1.

A major advantage of HEDL is the intuitive and powerful mechanism for data model extensions through composition. Thus, new persistence layers can be generated by reusing and extending existing domain models. Listing 2 shows the HEDL file of the component C_2', which adds further attributes to C_1.

```
extendModel="c1.hedl"
Article {
    Int value;
    String measure;
}
```

Listing 2: HEDL model for C_2.

The developer of C_2' directly works with a generated Java class for the `Article`, including the defined attributes. A Hibernate mapping is generated, which allows creating and accessing the database schema. DAVE's DB Development Tool takes the new schema, the previous version of the component, and all dependencies of the new version as input.

Output. The output is a representation of the delta between the previous and the current version of the component. The DB Development Tool adds the database folder to the component's source, consisting of three files and two subfolders. First, the file `dependencies.xml` collects the components and their versions, which are used or extended by the component project. Second, the `ini.liqui.xml` file is the Liquibase script which creates the component's schema from scratch or by extending existing dependencies. Third, the `evolve.liqui.xml` file contains Liquibase operations to transform the previous version of the component into the new one. Fourth, the `history` folder contains the database folders from all previous versions of the component. This is necessary to perform updates even on older versions than the previous one. Fifth and finally, the `sql` folder contains SQL scripts, which are linked from `ini.liqui.xml` and `evolve.liqui.xml`, describing the evolution of the data. To realize the evolution of existing data during deployment, the DB Development Tool generates SQL templates into the `sql` folder for each new column or table. The developer has to fill these templates manually. Nevertheless, the generated templates guide the developer through this task. Given the evolution to C_2', DAVE generates an update statement template for the new columns `value` and `measure` of the `Article` table. The developer can assume the old table `Article` to be still present.

Implementation. The algorithm to create the initialization file and the evolution file uses two databases (DB_{new} and DB_{ref}) and Liquibase's feature to compare given database schemas. To create the `ini.liqui.xml` file, three steps are necessary: First, we use the generated Hibernate mapping to create the component's database schema to DB_{new}. The extension mechanism of HEDL inherently initializes all dependencies. Second, we create the schema of a customer product, containing exclusively the component's dependencies, using their initialization scripts, to DB_{ref}. Finally, we use Liquibase to compare the two database schemas and retrieve the `ini.liqui.xml` file. To enrich this schema evolution with data evolution, we create SQL templates for every new column or table, store them in the `sql` folder, and link them from the Liquibase script. The tool generates an `UPDATE` statement for each table including new columns and an `INSERT` statement for each new table. The component developer has to use these templates to specify the new values depending on the old data, provided by the existing dependencies. For instance, the initialization script of component C_2' adds two columns to the table `Article` and generates templates for the corresponding update statements. After completing the SQL templates, the initialization script is finished and ready to use for any initial deployment of the component.

If there is a previous version of the component, the evolution is stored in the `evolve.liqui.xml` file. The DB Development Tool creates it by executing the following four steps: First, we create the schema of the new component to DB_{new} using its Hibernate mapping. Second, we initialize the previous version using the initialization script of the predecessor and the predecessor's dependencies to DB_{ref}. Third, we adjust the dependencies to match the new component version. This includes three possible scenarios: adding a dependency (`ini.liqui.xml`), removing a dependency (inverse of `ini.liqui.xml`), and updating a dependency (`evolve.liqui.xml`). Finally, we again use Liquibase to diff between DB_{new} and DB_{ref} to obtain the evolution script `evolve.liqui.xml` and generate the SQL templates for data evolution. When filling the generated SQL templates, the developer may assume the previous version to be still present.

Consider the evolution from C_2' to C_2'', in the generated evolution script, we add the two attributes to `General_Cargo`. We do not need to remove the previous columns added to `Article`, since this whole relation is dropped by the evolution of C_1 to C_1'. However, in the SQL statement templates we assumed the `Article` table to be still present and transform the data to the new version. This database evolution script is sufficient to execute any deployment including the database evolution between arbitrary configurations. Please note, that DAVE does not support evolution to predecessor versions of components.

4.2 DB Deployment Tool

After the development and the generation, a concrete customer product is ready to being deployed. If the customer already runs an older version of his product, the deployment has to keep the old data and transform it according to the new configuration. The concrete evolution script will be derived by DAVE's DB Deployment Tool from the generic description in the database folder of each component. As a consequence, the deployment of products and the development of single components are decoupled completely.

Input. The SPL contains components, including their created database folders. After the customer chooses the desired features from the feature model, the corresponding components are composed to the final product. The local database folders of these components are simply collected and serve as input for the deployment step. Another important input is the previous product and its configuration (the set of previously installed components and their version numbers). This is necessary to determine for each component whether it is evolved, added, removed, or stays unchanged.

Output. The DB Deployment Tool creates a global Liquibase script for the database evolution. It ensures the correct evolution of both schema and data of the currently installed product to the new one.

Implementation. To generate a correct database evolution script for a customer's deployment, the DB Deployment Tool considers the currently installed configuration and derives the necessary steps to obtain the new one. Figure 3 shows possible configurations according to the example in Figure 3. As an example, let us consider the product evolution from $\{C_1, C_2'\}$ to $\{C_1', C_2''\}$. Given the new and the previous configuration, DAVE determines the sets of added, removed, and updated components and collects the required Liquibase operations respectively. These operations originate either from the initialization script, its inverse, or the evolution script. In case a component's update skips versions, the tool also includes the corresponding evolution scripts from the `history` folder.

DAVE's DB Deployment Tool interleaves the collected database operations, since it is not feasible to simply execute the whole scripts sequentially. Evolution steps may influence each other. For instance, the evolution of C_1 to C_1' creates the tables `General_Cargo` and `Bulk_Cargo`, inserts the data from the table `Article` accordingly, and finally drops `Article`. The evolution of the additional component from C_2' to the version C_2'' adds the two columns to `General_Cargo` and inserts the data from `Article`. It can be applied neither before nor after the evolution of the component C_1. If the evolution to C_2'' is executed first, the table `General_Cargo` is not created yet and the addition of the new columns and the insertion of data would fail. If the evolution to C_2'' is executed last, the original `Article` table is already dropped including the data in the additional columns. This is, in its essence, the weaving problem.

The DB Deployment Tool solves the weaving problem with the help of operation groups. It groups database operations of the same kind across all components and arranges these groups sequentially. Mainly, there are seven phases in the resulting evolution script. First, the DB Deployment Tool executes all database operations that add information capacity to the schema, like (1) *creating tables* or (2) *adding columns*. Afterwards, (3) all obsolete *constraints are removed* to (4) execute *data evolution*. At this point, DML and DQL operations can access new schema elements and also the old ones. The previously existing data is still fully available and can be inserted into the also existing new structures. Finally, the DB Deployment Tool executes all database operations that reduce the information capacity, like (5) *removing columns* or (6) *dropping tables*, to obtain the desired schema. This also includes (7) *adding new constraints*.

Within an operation group, the DB Deployment Tool orders all operations according to the topological order of the original components regarding their dependencies. Multiple operations of one component in one group remain in the order specified by the developer. In our example, the previously installed product uses the `Article` table with the additional `value` and `measure` attributes. The final evolution script would start by (1) creating the new tables `General_Cargo` and `Bulk_Cargo`, (2) adding the `value` and `measure` attribute, and (4) inserting the data of C_1 into C_1' and updating the additional attributes of C_2'' using the original `Article` table. Afterwards, it (6) drops the `Article` table and (7) adds the not-null constraints to the new tables. This finally creates the desired schema including the transformed data.

5 RELATED WORK

While software product lines are an exhaustively studied subject in software engineering, database management issues in SPLs are underrepresented in research. According to the perception that a database consists of its schema and its data, we distinguish database schema evolution and accordant data evolution in SPLs. Both aspects of database evolution in SPLs are relevant for the consistent evolution of components and products as motivated in Section 3.

Variable database schemas in SPLs are studied in (Khedri and Khosravi, 2013) and (Abo Zaid and De Troyer, 2011). Modeling data variability in SPLs is typically based on feature modeling as used in SPL engineering. In (Abo Zaid and De Troyer, 2011), a *variable data model* is introduced. Before variability of *data concepts* in the variable data model can be defined, *persistency features* in the feature model of the SPL are specified by the extended Feature Assembly Modeling Technique (Abo Zaid et al., 2010). However, this technique only considers the initial derivation of a product's database schema.

The evolution of a database schema for an SPL product is analyzed in (Khedri and Khosravi, 2013). Delta-Oriented Programming is used to add delta modules, defined by SQL DDL statements, to a core module incrementally, based on the product configuration. Database constraints are generated for the delta scripts to ensure a valid global database schema.

To the best of our knowledge, there is no research on data evolution in software product lines. In our understanding of SPL evolution (cf. Figure 3), component evolution is closely related to database refactoring (Ambler and Sadalage, 2006). There is sufficient support for database evolution of one running product, like e.g. Liquibase or Rake. However, the SPL evolution, hence the weaving problem, still requires in-depth research. It requires the generation of global evolution scripts from the component's local scripts. Since the management of a software product line and the derivation of its products is mostly model-based, results from model-driven engineering research are relevant. In (Milovanovic and Milicev, 2013) it is reported about a pragmatic and efficient solution to the problem of schema evolution affecting existing programs, in the domain of model-driven development of database applications using Unified Modeling Language (The Object Management Group, 2010) (UML) models. The main contribution of this paper is a semi-automatic algorithm for differencing structural UML models and upgrading the relational schema, as well as a tool that has been evaluated in a large-scale e-government human resources management system.

6 CONCLUSIONS

In the eTe project, we laid our focus on an important but widely unstudied problem: database evolution in SPLs. SPLs decouple the development of the components from the actual deployment of products. The developer of a component specifies its local data model. According to a customer's requirements, such components are composed to a product. The global database schema of such a product is derived by composing the local schemas of all components.

Since evolution is inevitable, the customer's product will evolve, including the addition, removal, or update of components. The customer relies on a consistent database, which has to be evolved accordingly. Consequently, the database evolution during deployment requires to derive the specific global evolution script from given local definitions within the components. Creating a correct (logical) and efficient (physical) evolution script is called the weaving problem.

Obviously, this is a general problem, which is not restricted to the eTe scenario. To achieve a valuable general solution, we first discussed the general problem of (database) evolution in SPLs and formulated general challenges. We presented DAVE, a database evolution toolkit for the eTe SPL. It weaves the local evolution scripts to a global evolution script by grouping the single database operations into groups, which are then executed sequentially. Within each group, the operations follow the topological order according to defined dependencies between components. Within each component, the original order is kept.

DAVE solves the weaving problem on the logical level. We successfully tested DAVE on realistic evolution scenarios based on the detailed experience of the industry partners. It emerged as being capable of realizing database evolution for eTe. DAVE retains a lean runtime architecture, since DAVE does not introduce any additional layer at runtime for database evolution. Developers can rely on commonly known tools and technologies, which was an important requirement of the eTe project.

The concepts of DAVE are universal and applicable to SPLs in general. SPLs and their evolution are promising trends in software and database technology and we consider DAVE as an important contribution particularly because of its practical background.

ACKNOWLEDGEMENTS

We thank our partners of the eTe project. This research has been co-funded by the European Regional Development Fund in the project #100135681/2804.

REFERENCES

Abo Zaid, L. and De Troyer, O. (2011). Towards Modeling Data Variability in Software Product Lines. In *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg.

Abo Zaid, L., F., K., and De Troyer, O. (2010). Feature Assembly Modelling: A New Technique for Modelling Variable Software. In *5th International Conference on Software and Data Technologies Proceedings*, volume 1, pages 29 – 35. SciTePress.

Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In Obbink, H. and Pohl, K., editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin Heidelberg.

Chen, K., Zhang, W., Zhao, H., and Mei, H. (2005). An approach to constructing feature models based on requirements clustering. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 31–40.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

DevBoost (2013). HEDL - Hibernate Entity Definition Language (Hibernate DSL - User Guide).

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document.

Khedri, N. and Khosravi, R. (2013). Handling Database Schema Variability in Software Product Lines. In *Software Engineering Conference (20th APSEC*.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*.

Milovanovic, V. and Milicev, D. (2013). An interactive tool for UML class model evolution in database applications. *Software & Systems Modeling*, pages 1–23.

Murer, S., Bonati, B., and Furrer, F. J. (2010). *Managed Evolution: A Strategy for Very Large Information Systems*. Springer Berlin/Heidelberg.

Pohl, K., Böckle, G., and Van Der Linden, F. (2005). *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information & Software Technology*, 37(7):383–393.

Terwilliger, J. F., Cleve, A., and Curino, C. A. (2012). How clean is your sandbox? In *ICMT*, volume 7307.

The Object Management Group (2010). OMG Unified Modeling Language TM (OMG UML), Superstructure, Version 2.3.