Handling Database Schema Variability in Software Product Lines

Niloofar Khedri
School of Electrical and Computer Engineering
College of Engineering
University of Tehran, Tehran, Iran
nkhedri@ut.ac.ir

Ramtin Khosravi
School of Electrical and Computer Engineering
College of Engineering
University of Tehran, Tehran, Iran
r.khosravi@ut.ac.ir

***Abstract*—Managing variability in a software family is crucial to software product line engineering. The existing variability management techniques, however do not particularly address database design in the context of information systems product lines. This paper presents a practical approach to handle variability in database design for families of software. We use the technique of Delta-Oriented Programming when a product is constructed by adding a number of delta modules to a core module incrementally, based on the features selected in the product configuration. We use SQL Data Definition Language to model core and delta modules. We present rules for consistency checking of the delta scripts based on the database consistency constraints to generate a valid consistent database schema for the product. Also we analyze the cases in which a conflict arises based on inconsistencies between delta modules. The fact that DDL is widely known to software developers, along with modularity and scalability of the proposed method makes it suitable to be used in industrial real world applications.**

***Keywords*-Software Product Line Engineering, Data Model Variability, Delta-Oriented Programming**

I. Introduction

Software product line engineering is an approach to manage diversity in families of software products. In software product line engineering, the commonalities and variabilities of the products are identified and a reusable platform is constructed to decrease the time to market and the cost of the product in average. Software engineering process is divided into domain engineering and application engineering processes. In domain engineering, we define the commonalities and variabilities of the products and establish a reusable platform [1]. A large group of variability management techniques are categorized as feature-oriented techniques like FODA [2]. Feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software [3]. A feature can be decomposed into several mandatory, optional, or alternative sub-features and features can have “or”,“requires” and “excludes” relationships [1]. A feature model contains the features and feature relationships. In application engineering, the product line application is derived from the established platform.

As data plays a fundamental role in software intensive systems and information systems in particular, variability modeling in software intensive product lines should address the diversity in data and data models. Moreover, the variability in database is not necessarily identical to that in application. There are few studies in the area of software product lines (SPL) addressing the variability in enterprise information systems [4], but the need for variability management in these systems is no less than other types of systems. The mentioned approaches have not been particularly designed to support data model variability. In addition there are some studies on data model variability [5], [6], but they are in the data model or domain model level and do not directly address the design and implementation of the database. As a result, an appropriate variability modeling approach supporting variability at database design level is necessary.

The growth of object oriented analysis and design methods encourages software developers to start from an object-oriented domain model and then use object relation (OR) mapping frameworks to generate the data model [7]. The issue of whether the data model must be designed separately, or must be the result of automatic code generation based on an object-oriented data model is still a debate among practitioners in software development [8], [9]. The “impedance mismatch” issue may lead to serious performance problems when using OR mappers. Separating the data source layer (which includes the database) from the domain layer and presentation layer in order to have independent layers and hide the complexity and details of each layer from others encourages us to present a method for representing and managing variability in data model of software intensive systems.

In this paper, we present a new variability modeling method suitable for handling variability in data model at the database level. Our approach is based on the delta-oriented programming (DOP) technique. In DOP, a product is generated by adding delta modules to a core module incrementally [10]. The core module consists of the mandatory features of the product family and delta modules are related to the alternative or optional features. As DOP is not restricted to any particular modeling or programming language, we can apply it to database programming languages in order to implement the product data model. We generate the core data model from the mandatory and some selected alternative and optional features and then build a delta module for each feature outside the core as a modification to the core module. We have chosen Data Definition Language (DDL) of SQL as a well-defined and well-known language to describe core and delta modules. According to the DOP technique, the data model of a product is generated automatically by adding the deltas related to selected features of a given product configuration to the core.

In order to automatically generate a data model for a valid product configuration, inconsistencies in delta modules must be identified prior to the development of the product data model. Our method to check the consistency of the delta modules is based on the database consistency constraints.

We claim our proposed method to be practical in the following aspects:

- As mentioned, we use DDL as the description language for the core and delta modules widely used by the practitioners. As the method does not require a special tool, practitioners' learning curve is relatively flat.
- Most parts of the processing needed can be performed by the database management systems; hence there is little need to develop new tools.
- The fact that we model the variabilities in separate delta modules makes our method scalable in practice, as opposed to the methods that try to represent all variabilities in a single model.

We choose a family of university software systems as the case study. For the case study, 6 web-based university software systems in our country are studied. One of the systems has 4 different versions deployed in 4 different universities with configurable features. Note that, none of these applications has been developed based on software product line engineering. This case study is part of a project currently being done in the Software Architecture Laboratory in University of Tehran on developing a mid-size product line for university information systems aimed at discovering and addressing problems in development of information systems product lines.

II. Related Work

The approaches to manage diversity in database design of software product line can be classified in two main directions: database context, SPL and model merging context.

The related research in database community includes schema integration [11] and reuse and refactoring techniques in database [12], [13], but they express their methods without targeting explicit variability management.

In the context of SPL, some studies propose to represent data model variability. ADMV process (Addressing the Data Model Variability) is a UML-based approach for SPL data modeling and data integration. [5], [6] ADMV uses adapters and views in order to support the data integration and variability in SPL. It is important that [5] and [6] do not consider the logical and conceptual modeling of the database and apply their proposed technique on a UML-based domain model.

An approach for modeling data variability is provided as part of the overall software product line modeling approach [14]. The authors present a method to represent data model variabilities in a single data model. As the size of the product family increases, representing the data model variability in a single model is not a suitable way to manage variability in data model and the resulting data model can be very complicated. Instead, our proposed method uses DOP as a modular approach to handle variability in database design. As each feature is expressed in terms of a separate set of DDL statements, we will have a set of relatively simple models.

Also, [15] focuses on view integration to generate a global consistent schema out of different local schemas for users or views. Two solutions based on decomposition are presented to manage variability on the level of conceptual modeling. The mechanism to compose a schema of a product is superimposition. Superimposition is a technique to merge code (or model) pieces belonging to different features. This proposed approach is similar to our method in decomposing database schema according to feature model. The authors apply their methods on ER model and we apply our method on SQL DDL scripts in order to implement a database schema. There are two main reasons to apply DOP on SQL DDL scripts in our proposed method. First, it is easier to use textual languages (like SQL) in DOP than graphical languages like ER, because merging fragmented ER diagrams is hard. Second, in the case of applying DOP on ER, we should present a new notation in order to represent changes to ER diagram (for example, how an entity can be removed from an ER diagram). It is important to point out that the devised notation may be not familiar for the developers but almost all database developers are familiar with DDL, making our approach practical to use. As our proposed method is based on delta-oriented idea, we present a method for applying the deltas and check the consistency rules of the delta scripts.

The superimposition technique for UML models is used in [16] in order to provide a tool for UML model composition with variability support. They apply the approach on UML class diagram, state diagram and sequence diagram to model variability. These studies concentrate on managing variability in general, and not in data model. Also as mentioned before, our approach is at the database level.

An approach is proposed in [18] on relating variability model and design model using UML package merge mechanism. Based on the variability model and using model driven techniques, the package architecture and the product configuration are automatically produced. The approach focuses on package merge operator provided by UML. In our work, we assume to have both merge and remove operators to build a concrete product, because of the fact that using only merge operator makes us use a simple minimal core consisting of

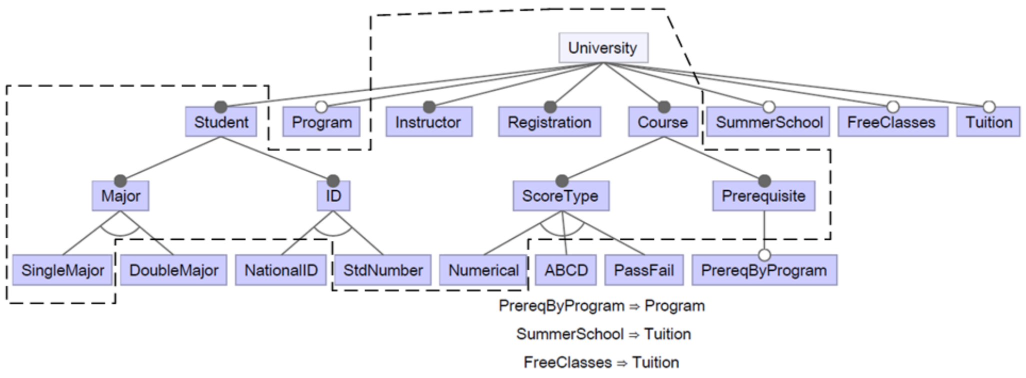

Figure 1. Simplified feature model - set of features in dashed line represent the core

mandatory features. Deltas are added to the minimal core in order to construct more complex products by merging deltas to the core. Instead, if we have negative delta scripts, we can include the features that are present in most (but not all) configurations, and remove them in the few products excluding those features.

III. Delta-Oriented Programming in Database

Delta-oriented programming approach is presented in [10] and [20], as a way to implement SPLs. In DOP, the implementation of a product family is divided into a core module and a set of delta modules [10]. One can implement a product by starting from a core module and then apply delta modules to the core in order to implement different features. Delta modules can add (remove) code to (from) the product. In the context of database programming, a core module contains a set of DDL scripts, which includes tables, columns, data types, primary key and foreign key references (collectively referred to as database objects). In [10], the core module implements a complete product for a valid feature configuration. We can implement a maximal core which includes all possible elements of the data model or a minimal core which only contains the common mandatory features. Here, we assume that the core module contains at least mandatory features and a set of selected alternative and optional features. Alternative and optional features are selected manually based on the frequency of the feature in family. Therefore, the core consists of DDL scripts for creating database objects for mandatory features and the selected alternative and optional features.

As delta modules specify changes to the core module in order to implement the features outside the core, the delta scripts change the core by creating new tables, dropping existing ones and modifying the existing tables via adding, modifying and deleting columns, data types, primary key and foreign key references.

Figure 1 illustrates our running example of a simplified university product line feature model. We ignore the parts of the feature model not affecting the data model. As illustrated in Figure 1, *PrereqByProgram* feature *requires Program* feature, meaning that if product *P* has *PrereqByProgram* feature, it must have *Program* feature too.

In the core (Figure 1 - set of features in dashed lines), we assume that the student identity is *StdNumber* and each student studies in only one major (*SingleMajor* feature). Also, only *Numerical* score type is supported. The core script in university product line should consist of DDL script for creating entity and relationship tables of the features in the core (illustrated in Figure 1).

In the core, each course has some *prerequisite* courses. However, if we choose *PrereqByProgram* feature, the prerequisites are specified for each program. For example, the course *C1* may have *C2* as its prerequisite in a program *P1*, but not in another program *P2* (Figure 2). *PrereqByProgram* delta drops the table related to *prerequisite* relation (of the *Course* entity) and creates *Program*, *Requirement* and *Prerequisite* (for *Requirement*) tables and alters the *student* table to add the relevant information.

In the selected core, a student studies in only one major. Having the *DoubleMajor* feature, each student can study in two majors. Note that this feature can be implemented in two ways. First, it can be applied by just adding a column to the *Student* table (named *Second Major*) and second, one can implement it by adding a separate relation (effectively modeling an n-to-m relation between *Student* and *Major*).

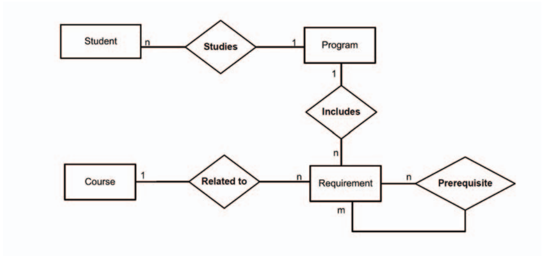

Figure 2. PrereqByProgram ERD

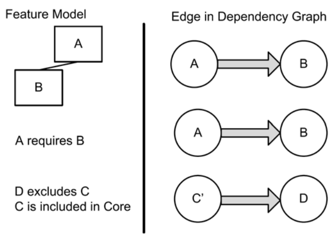

Figure 3. Feature dependencies in graph

IV. Data Model Generation Based on Configurations

Studies exist on checking the consistency of the feature model and product configuration as [21]. In our method, delta scripts may add, remove or modify the core, so we have to apply the delta scripts sequentially (in a proper order) to avoid conflicting modifications (e.g. adding an attribute to an entity which is not added yet into the core). The way to find a proper sequence to apply deltas is to create a dependency graph for the selected features and then apply topological sort algorithm to find the sequence of applying the features. The dependency graph related to a set of features is created as below:

1) When feature *A* is part of the core and it is not selected in product configuration, there is a node *A'* in dependency graph (*A'* represents removing *A* from the core). As a result, there should be set of deltas that each removes a specific (non-mandatory) feature from the core.
2) When feature *A* is not a part of the core but is selected in product configuration, there is a node *A* in the dependency graph.
3) When feature *B* requires feature *A* or feature *A* is the parent of feature *B* and feature *A* is not in the core, there is an edge from node *A* to node *B* (Figure 3).
4) When feature *D* has an alternative or excludes relation with another feature *C* and feature *C* is a part of the core, there is an edge from node *C'* to node *D* which represent the script to remove *C* feature from the core (Figure 3).

For example, suppose *DoubleMajor* (instead of *SingleMajor*), *NationalId* (instead of *StdNumber*), *ABCD* (instead of *Numerical*), *PrereqByProgram* and *Program* are selected as the product configuration. To generate the data model, the deltas related to *DoubleMajor*, *NationalId*, *ABCD*, *PrereqByProgram* and *Program* must be added to the core. *PrereqByProgram* requires *Program*. *DoubleMajor* is an alternative for *SingleMajor* (which is a part of the core), *NationalId* is an alternative to *StdNumber* and *ABCD* is an alternative to *Numerical* (The core includes *StdNumber* and *Numerical*, depicted in Figure 1). The sequence of applying deltas to the core is the output of applying topological sort to

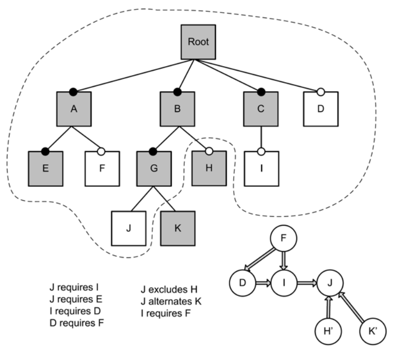

Figure 4. Top:Feature model and selected features - Bottom: Dependency graph

the dependency graph. There can be more than one ordering for deltas and one solution applies the deltas in the order: *StdNumber'*, *NationalId*, *Numerical'*, *ABCD*, *Program* and *PrereqByProgram*. When two features are independent there is no preference in applying related deltas, because feature composition is not generally commutative and the sequence of applying deltas must obey the dependencies among the related features.

Usually the dependency graph is a directed acyclic graph. Having an unstructured feature model which represents a cycle is rare in practise. In this paper, we suppose to have a structured feature model. For instance, suppose the sample feature model shown in Figure 4. Features with gray background are the core features and the set of features in dashed line represent the selected configuration. The complete dependency graph of the selected product is depicted in Figure 4. One possible topological sort of the dependency graph is: *F*, *D*, *I*, *H'*, *K'*, and *J*.

V. Consistency Checking of DDL Scripts

The automatic generation of the product data model by delta application is only performed if the desired product line configuration is a well-formed one [10]. In the context of database programming, this means that all the delta DDL scripts of a data model product must be well-formed too.

A delta DDL script can be added to the core if all the database objects dropped or changed exist and the ones created do not exist. Our method to check the consistency of the delta DDL scripts is based on the database consistency constraints. Inconsistencies in data model must be identified prior to the development of the product data model.

1) Existence rules: In some DDL operations, we assume that a database object exists and then we modify (Alter) or delete (Drop) it. For example, in Drop Table A, it checks that a table with name A exists.
2) Non-Existence rules: In some DDL operations, we assume that an object is not in the database when we create it.

Table I
DOUBLEMAJOR DELTA SCRIPT

```
CREATE TABLE Major( MajorID int NOT NULL,
    name nchar(10) NOT NULL,
    CONSTRAINT PK_Major PRIMARY KEY
    CLUSTERED ( MajorID ASC))
CREATE TABLE StdMajor( StdId int NOT NULL,
    MajorId int NOT NULL,
    CONSTRAINT PK_StdMajor PRIMARY KEY
    CLUSTERED( StdId ASC, MajorId ASC ))
ALTER TABLE StdMajor WITH CHECK ADD CONSTRAINT
    FK_StdMajor_Major FOREIGN KEY(MajorId)
    REFERENCES Major (MajorID)
ALTER TABLE StdMajor CHECK CONSTRAINT FK_StdMajor_Major
ALTER TABLE StdMajor WITH CHECK ADD CONSTRAINT
    FK_StdMajor_Student FOREIGN KEY(StdId)
    REFERENCES Student (StdId)
ALTER TABLE StdMajor CHECK CONSTRAINT FK_StdMajor_Student
```

3) Reference rules: In some cases, it is necessary to check the references to a primary key.

For instance, in the delta related to *DoubleMajor* feature (Table I), considering *Non-Existence rules*, *Major* table and *StdMajor* table must not exist and then, existence of the *MajorId* primary key in *Major* table is checked when the foreign keys for *StdMajor* table are defined.

Morover, we need to modify a table several times in a DDL delta script in some cases; thus, for every table modified more than once in a delta script, database objects are different or the delta DDL scripts conform to the priority rules as follows:

1) If there are *alter column*, *add constraint-primary key* and *add constraint-foreign key* statements on a specific column in a set of delta statements, *alter column* statement(s) should be executed before *add constraint* statement(s).
2) If there are *alter column*, *drop constraint-primary key*, *drop constraint-foreign key* statements on a specific column in a set of delta statements, *drop constraint* statements have higher priority than the *alter column* statement(s).

For instance, if the *NationalId* feature is selected, in the related delta DDL the national identity number is added to *Student* table as primary key, so all the previous references to *StdNumber* must change their target reference to *NationalId*.

In order to check the consistency of the delta DDL scripts, the scripts are checked according to the SQL rules. Further, if the statements of a script work on the same database object (e.g., the same table), the target object should be different (e.g., two different fields of that table) or they should conform to the priority rules. Although it is good to have a special tool for checking the consistency, this is not essential because the database itself checks all these rules.

VI. Discussion on Feature Interactions

In this section, the consistency between delta DDL scripts is discussed. In some cases, two delta DDL scripts work on the same part of database object (for example, they add the same column to a table), as a result database management system stops executing the scripts, resulting in an SQL error. So, when two features have interaction with each other in a way that they both modify the same part of the database object, the process of generating database model is stopped. According to [22] when two deltas are in conflict, they are incompatible and no ordering can solve the problem. The conflict can be handled by analyzing the delta DDL scripts of the related features and then derive a correct delta DDL script for the case of selecting both features together [22]. Then nodes related to conflicting deltas in dependency graph are replaced by a new node, representing both features and find a new sequence for applying deltas.

For instance, consider a university that admits students for summer school and a university that holds free classes. The delta DDL scripts related to each of the optional features, *SummerSchool* and *FreeClasses*, add a new discriminator column *Type* to *Student* table and also add their own tables to implement *IS-A* relationship. When both features are selected in a product configuration, the delta DDL script related to second feature (which is added to the core) again adds a discriminator column with the same name to *Student* table, causing an inconsistency between the scripts. We can create a new delta DDL script and apply it on the core when these two features are selected together.

Sometimes, the database object property (e.g. data type for columns) in the delta DDL scripts are not the same and the inconsistency cannot be easily resolved. Generally in some cases, the inconsistency problem could not be solved easily due to data inconsistency between delta DDL scripts or database design problems; therefore the inconsistency problem is reported to database designers, who can resolve the issue manually.

Finally, the generated database should be checked so that it has meaningful objects, meaning that all tables must have at least one attribute and all relations must be related to at least one entity. So a process (post-process) required to eliminate any table without attribute and relations without entities.

Usually, cross-cutting features make similar changes to several places in the model. In database context, a cross-cutting feature like attaching a "last updated" timestamp to every entity is implemented by adding the *Timestamp* column to all tables. The proposed method does not support crosscutting features, because the related delta DDL script needs information about all the tables. A meta-data table or catalogue of the whole database is required and all delta DDL scripts should update meta-data too.

As a whole, we can conclude the process of generating database as below:

1) Build the core module for mandatory and selected alternative and optional features.
2) Build delta modules for each (non-core) optional and

alternative feature. Also, build delta modules for each optional and alternative feature in core to remove it from core.
3) Create dependency graph based on the selected product configuration (selected features which are not part of the core and features in core which is not selected).
4) Apply topological sort to dependency graph. (There is at least one sequence for any well-formed feature configuration)
5) Apply deltas on the core according to one of the results of topological sort.
6) If there is no conflict, go to step 10.
7) If there is any conflict between *Delta A* and *Delta B*, analyze both scripts and derive a new delta *Delta AB*. Note that, if there is no solution to handle the conflict (it may happen because of data inconsistency or database design faults), ask database designers to solve the problem.
8) Replace *AB* in dependency graph wherever node *A* and node *B* exists.
9) Go to step 4.
10) Apply post-processing to remove tables without any attributes and relationships without any entity.

In [19], main challenges of merge operator on database models are studied. Here, we discuss these challenges and their applicability on presented method. The first challenge *"the need for synchronization"* studied the problem of feature interaction and updating the data model in large applications. In our presented method, feature interaction is handled through a separate delta DDL script for any set of interacting features. Competitive features that add the same database object element with different properties are not allowed. Also, we check for the errors and conflicts of applying each delta to the core to detect competitive features, feature interactions, and problems. As the presented method builds the dependency graph and applies topological sort in the case that the conflict is handled by a new delta, further conflicts can be recognized.

The second challenge *"variability notation in base diagram"* is about managing variability in design level, for instance showing the variability in relation cardinality in data model. In our proposed method, we manage variabilities with delta DDL scripts and we can handle this kind of variability with assigning the related delta DDL variability to the feature.

The third challenge is *"relation between fragments and features"*. The proposed method handles the mapping of features (which do not participate in the core) to schema elements directly due to the fact that each feature is generally implemented in a delta DDL script by using the common database implementation techniques. When two features have common elements and modify the same part of the data model, a separate delta DDL script handles the set of interacting features. So each schema element can be annotated with the related feature(s). As a result, mapping schema elements to features is done backward by annotated data attached to schema elements.

VII. Case Study

A family of university software systems is chosen as the case study, because the research group consists of university students familiar to this application domain as well as the possibility to study a group of such systems.

The method for analyzing the case study is based on [23]. A group of software engineering students have studied 6 university software systems in our country. One of the systems has 4 different versions deployed in 4 different universities. All of the studied applications are web-based and none of them has been developed based on software product line engineering, only the one that is used in 4 universities has configurable features. We analyze university software systems by using the system, interviewing the users and domain experts. Features are extracted manually based on the commonalities and variations in the studied systems and organized into a feature model. The features were extracted based on the functionality of the systems observable through their user interfaces (via different system roles). The features common to all products were considered mandatory. The feature model was reviewed by another team member for possible inconsistencies. Also, an external domain expert further reviewed the final feature model. The set of mandatory features along with other features used more frequently in the products were selected as the core features.

In parallel to feature modeling, the domain was analyzed using object-oriented analysis method of [24]. As a result, a domain model was expressed in a number of UML class diagrams corresponding to the core and the delta modules. These diagrams were used to design data models and DDL scripts for each module.

We have selected 26 out of 91 features in the whole family as the case study (depicted in Figure 5). The core contains 8 mandatory and 4 selected optional and alternative features. The set of features enclosed in the dashed line represents the core. We build the core in only one DDL script, building 8 tables. There are 9 optional and 5 alternative features which are added to the core by applying a single delta script per feature. The consistency of all delta script is checked before, deltas are added to the core according to the sequence derived from applying topological sort to feature dependency graph.

Although there exist thousands of valid configurations according to the feature model of Figure 5, we chose 8 products which especially include *Alternative* features (not in core) and *Require* relations.

As the goal of this research is to provide a practically effective method to manage variability in the data models. We performed the mentioned case study to check whether

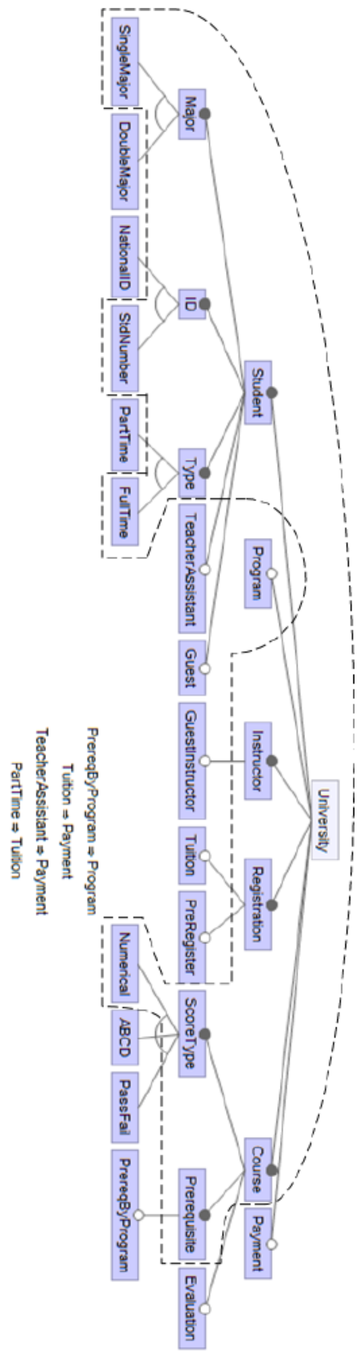

Figure 5. The selected part of the feature model - the part enclosed in the dashed line represents the core

the method can be applied to a real mid-size project without significant overhead. The results of the study confirm that the learning curve is relatively flat and the needed effort is reasonable compared to doing a project of a similar size without incorporating variability.

In our method, the database management system is responsible for detecting the conflicts and the conflicts are resolved by the delta module designer. It takes time and effort to trace back the error and find the root of the conflict, because in some cases the database error message does not directly indicate the reason of the conflict. Also, the database designer must review the previously applied deltas to find the conflicting delta(s). The fact that we have not used any special tool developed to be used in our method has the benefit of making our method easy to learn, but at the same time, debugging delta modules may become harder.

It takes time for a delta module designer or a database designer to find out the real reason behind the conflict. This problem can happen in the methods assumed to have a single model to represent data model variability too as [14].

Although the textual syntax of SQL DDL scripts helps us define and manage delta modules easily, when the size of the product family increases, it is hard to understand and manage delta modules. As a result, it is required to have intermediate ER models and build delta modules based on them.

VIII. Conclusion and future work

In this paper, we tackled the data model variability problem in SPL, not addressed before at the database level. We introduced a practical method for implementing the database schema particularly designed for SPL in a way that the variability is applied to the database design systematically. Our approach relies on delta-oriented programming technique. The data model of a product is generated by applying delta scripts to the core scripts sequentially based on sequence derived from applying topological sort to dependency graph of the selected features.

An advantage of applying DOP technique in database context is that the ordering of deltas can be determined based on the selected features. This is not the case in the original DOP method, where each delta module can have an "After" clause (indicating the order of adding the deltas to the core), meaning that the programmer himself should define the order of the deltas.

As a part of this work, we presented the rules for validating the consistency of the delta scripts according to the database consistency constraints. As a result, our approach enables us to evaluate the correctness of the deltas scripts and generated products. The inconsistencies of the deltas are detected at the configuration time; we plan to solve the inconsistencies between deltas at the design time. The proposed method was applied on the family of university software systems. As the method is not dependent on a specific domain, we plan to apply it on another domain (insurance software systems) to further investigate the generality of the proposed method. Also, we plan to extend our method to support cross-cutting features in future.

Furthermore, it may be important to analyze the cost of queries in the system, the result may also affect the decision between various implementations in both core and delta modules.

Acknowledgment

This research was in part supported by a grant form Informatics Services Corporation. (http://www.isc.co.ir)

References

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

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, Carnegie-Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *Proc. of the 11th Int. SPLC*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34.

[4] Y. Ishida, "Software product lines approach in enterprise system development," in *Proc. of the 11th Int. SPLC*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 44–53.

[5] J. Bartholdt, R. Oberhauser, and A. Rytina, "An approach to addressing entity model variability within software product lines," in *Proc. of The 3rd Int. Conf. on Software Engineering Advances*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 465–471.

[6] J. Bartholdt, R. Oberhauser, and A. Rytina, "Addressing data model variability and data integration within software product lines," *Int. Journal On Advances in Software*, vol. 2, pp. 84–100, 2009.

[7] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[8] T. Neward, "Ted neward's technical blog," Weblog, 2006, (December 1, 2012). [Online]. Available: http://blogs.tedneward.com/

[9] J. Atwood, "Is orm still the "vietnam of computer science?" Website, 2008, (December 1, 2012). [Online]. Available: http://stackoverflow.com/questions/404083/is-orm-still-the-vietnam-of-computer-science

[10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *SPLC*, 2010, pp. 77–91.

[11] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys*, vol. 18, no. 4, pp. 323–364, December 1986.

[12] M. Klettke, "Reuse of database design decisions." in *Proc. of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, ser. Lecture Notes in Computer Science, vol. 1727. Springer-Verlag, 1999, pp. 213–224.

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

[14] L. A. Zaid and O. D. Troyer, "Towards modeling data variability in software product lines," in *BMMDS/EMMSAD, Enterprise, Business-Process and Information Systems Modeling - 12th Int. Conf., BPMDS 2011*, vol. 81. Springer, 2011, pp. 453–467.

[15] N. Siegmund, C. Kastner, M. Rosenmller, F. Heidenreich, S. Apel, and G. Saake, "Bridging the gap between variability in client application and database schema," in *Datenbanksysteme in Bro, Technik und Wissenschaft(German Database Conf.)*, 2009, pp. 297–306.

[16] S. Apel, F. J, S. Trujillo, and C. Kastner, "Model superimposition in software product lines," in *In Proc. of the Int. Conf. on Model Transformation (ICMT), volume 5563 of LNCS*. Springer-Verlag, 2009, pp. 4–19.

[17] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jezequel, "Reconciling automation and flexibility in product derivation," in *Proc. of the 2008 12th Inter. SPLC*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 339–348.

[18] M. A. Laguna and J. M. Marques, "UML support for designing software product lines: The package merge mechanism," *Journal of Universal Computer Science*, vol. 16, no. 17, pp. 2313–2332, sep 2010.

[19] G. Saval, J. P. Puissant, P. Heymans, and T. Mens, "Some challenges of feature-based merging of class diagrams," in *VaMoS*, ser. ICB Research Report, D. Benavides, A. Metzger, and U. W. Eisenecker, Eds., vol. 29. Universität Duisburg-Essen, 2009, pp. 127–136.

[20] I. Schaefer, A. Worret, and A. Poetzsch-Heffter, "A model-based framework for automated product derivation," in *Int. Workshop on Model-driven Approaches in Software Product Line Engineering*, ser. MAPLE 2009, 2009.

[21] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes, "Automated diagnosis of feature model configurations," *The Journal of Systems and Software*, vol. 83, no. 7, pp. 1094–1107, 2010.

[22] D. Clarke, M. Helvensteijn, and I. Schaefer, "Abstract delta modeling," in *GPCE*, E. Visser and J. Järvi, Eds. ACM, 2010, pp. 13–22.

[23] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.

[24] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Addison Wesley Professional, October 2004.