Towards Modeling Data Variability in Software Product Lines

Lamia Abo Zaid[1], and Olga De Troyer[1]

[1] WISE Lab, Computer Science Department
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, 1050 Brussel
Belgium
{Lamia.Abo.Zaid, Olga.DeTroyer}@vub.ac.be, http://wise.vub.ac.be/

Abstract. In this paper, we provide an approach for modeling data variability as part of the overall software product line modeling approach. Modeling data variability in software product lines allows tailoring the data to the variability of a product. For this purpose, we have extended our Feature Assembly Modeling technique with the concept of *persistency feature*. We explain how these persistency features can be used to express the data variability, how they can be created and how they relate to the other features of the software product line. We also show how to derive a so-called *variable data model* from these persistency features and how an actual data model for a product of the product line can be derived. Additionally, annotations provide traceability between the variability of the features and the variability in the data model.

Keywords: Data intensive SPLs, Variable Data Model, Database Variability

1. Introduction

Software product lines [1] allow organizations to deliver different variants of a product to different customers in a structured way. Software Product lines are based on the idea of creating different products from a set of shared features. These features can be composed differently to create different products. Some features are common to all products, while others exist only in some of the products (i.e. if the functionality they provide is required by that product).

Software product lines have found their way in many application domains, for example: eHealth systems [2], Mobile phones [3], Revenue Acquisition Management solutions [4], and Web portals [5]. In all these examples, the product line deals with data one way or another: processing data, generating and storing data, or simply retrieving data. To differentiate software product lines that deal with a large amount of persistent data and those that don't, we will call the former *data intensive product lines.* Although much attention has been given to specifying software features and their variability, very little attention has been given to how the variability in product lines affect the variability of the data.

Development of efficient data intensive software product lines requires an alignment between the features of an individual product and the data on which these features operate. Different features may require different parts of the data. For different reasons, such as reducing the complexity of queries, increasing performance, or ensuring security, data that is not needed by the features of a certain product could be labeled to be discarded in some way or the other in the final product. For instance, different features selected for a certain product may require different views on a global database or different customized databases may be required for different products. This motivates the need to provide variability specifications at the data level and to have a link between the features of a product line and the data associated with it. In this paper, we will focus on data-intensive software product lines that maintain their data in a database. We call a data model (or schema) that supports variability a *variable data model (or schema)*. In this paper we propose an approach for modeling data variability as part of the overall modeling of the software product line variability. The proposed approach is based on extending our Feature Assembly Modeling technique [6] with so-called *persistency features*. Persistency features are features that represent persistent data within the application domain. We also show how the corresponding variable data model can be defined from these persistency features.

This paper is organized as follows: in section 2, we provide a brief overview of related work. In section 3, we present our approach for modeling data variability in software product lines. We demonstrate the approach on an example. Next, in section 4, we show how this information can be translated to the corresponding variable data model. In section 5 we show how this variable data model can be used to derive databases tailored to the needs of the different products of the software product line. Finally, section 6 provides the conclusion and future work.

2. Related Work

Tailoring the database to the specific needs of a database actor is a well-known issue in database design. Often many views are created to suit the specific needs of different users or user groups (i.e. actors). For example, in [7] *schema tailoring* was proposed to meet the needs of different actors accessing different portions of data, in different usage scenarios. Data views were tailored for different actors in different contexts. It amounts to cutting out the appropriate data portion that fits each possible actor-context.

In [8] a component based technique for creating Real Time Database Management System (RTDBMS) was proposed to help resolve the complexity of creating different types of automotive control systems. An aspect-oriented methodology was adopted to relate the components to the functionality provided by them. The proposed platform consists of a library of components and aspects, and is supported by a tool suite. The tool suit assists system designers in configuring and analyzing different configurations based on the specific requirements of the targeted automotive system.

The issue of matching the database with each member of the product line was first raised in embedded systems [9] [10] [11], where the hardware is diverse and only limited resources exist. Therefore it is very important that the application and its

accompanying database, as well as the database management system are suitably tailored to meet the different requirements. In [9] the tailoring process was done at runtime to provide a configurable real-time database platform (database and DBMS). In [10], a product line approach was adopted to develop both the application and the suitable database management system (and also data) for each product. In that case, it was crucial that with each product of the product line only the essential data management requirements and essential data existed. The focus was given to the variability of the DBMS features. The authors did not mention how the database entities were affected by this variability in DBMS features. In [11], a feature oriented programming approach for tailoring data management for embedded systems was proposed in which a feature model describing the DBMS features and their variability was created. Feature oriented programming was used to create a common architecture and code base that allowed to configure different configurations of the DBMS (the approach was applied to Berkeley DB).

In [12], the authors reported the need for a variable database schema to serve the different needs of the product line. Neglecting this need leads to a gap between the application and the database. The authors proposed to tailor database schemas according to user requirements. Two methods were proposed, *physically decomposed schemas* (i.e. physical views) and *virtual decomposed schemas* (i.e. virtual views) for representing variability in the application and matching this variability with variability in the corresponding database. Once a product is configured (i.e. the features of the product are identified) the schema is tailored to meet the needs of the product features. The proposed technique decomposes an existing database schema in terms of features. It allows tracing of the schema elements to the program features at the code level using a technique similar to the #ifdef statements of the C preprocessor.

In this paper we present an approach that allows modeling the variability of the data according to the variability of the application. Variability of the application is analyzed with respect to its influence on the corresponding data. Next, a variable data model is created which takes into consideration the need for tailoring the database based on the member features of each product at an early stage (i.e. data modeling stage).

3. Extending the Feature Assembly Modeling Technique to Model Data Variability

Feature oriented domain analysis [13] is used to analyze and model the capabilities of a software product line in terms of *features.* A feature represents a distinctive logical unit that represents a functionality, capability, or characteristic of the software. The product line is represented by a *feature model*. A feature model specifies features, their relationships, their dependencies, and how they relate to the variability of the product line. In the last two decades several feature modeling methods were defined for example: Feature-Oriented Reuse Method (FORM) [14], FeatureRSEB [15], Product Line Use case modeling for Systems and Software engineering (PLUSS) [16], Cardinality Based Feature Modeling (CBFM) [17], and Feature Assembly Modeling (FAM) [6].

In this paper we extend Feature Assembly Modeling (FAM) technique [6] to model variability for data intensive software product lines. Feature Assembly Modeling is a multi-perspective feature-oriented modeling approach. The multi-perspective approach adopted allows specifying software product lines from different perspectives. Each perspective considers one point of view to describe the variability of the software product line (e.g., the *System* perspective describes the variability from a general system point of view, the *Functional* perspective describes the variability from a functional point of view, the Graphical *User Interface* perspective describes the variability from the viewpoint of the Graphical User Interface). The purpose of using perspectives is to simplify the design process by adopting the principle of "*separation of concerns*". Each perspective only concentrates of the features and the variability relevant for that perspective. Note that FAM provides a variable and extensible set of perspectives. If a (pre-defined) perspective is not relevant for a given software product line it should not be used. Also new perspectives can be defined if necessary for a given software product line.
To allow modeling data intensive product lines, we have introduced a special perspective, called the *Persistency Perspective*. The persistency perspective focuses on describing the variability from the viewpoint of the persistent data. The features in this perspective, called *persistency features*, are defined from the point of view of their need for manipulating (creating, updating, deleting, querying) persistent data required by the different members of the product line. In principle, the feature models for the different perspectives considered are defined (modeled) independently. However, the feature model for the persistency perspective is derived from the feature models of the other perspectives, i.e. the persistency features are derived by inspecting the features in the other non-persistency perspectives modeling the product line. If a feature needs persistent data, a corresponding persistency feature is defined. Next, the persistency perspective model is completed by adding dependencies and relations between features and the consistency and completeness of the overall model should be validated. We will discuss each step in more details in the next sections. We will use a running example: a Quiz Product Line (QPL) application [6], which is variable software for making Quizzes and designed to meet the needs of multiple customers and markets. A Quiz application is mandatory composed of a set of system features namely: *Question*, *Layout*, *License*, *Report* Generator, *Operation Mode* and *Question Editor*. When applying the Feature Assembly Modeling methodology to QPL, the following perspectives were defined: a system perspective, a functional perspective, a user perspective and a graphical user interface perspective[1]. In this paper we focus on the persistency perspective.

3.1. Defining the Persistency Perspective

As indicated, the features in the persistency perspective are derived from features defined in the various other perspectives. For the QPL example these perspectives are: the system perspective, the functional perspective and the graphical user interface perspective. Each perspective contains features that might or might not be associated

[1] For more information on these perspectives please refer to [6] [18].

with persistency needs. If a feature has needs in terms of persistent data, a corresponding persistency feature should be created in the persistency perspective (if it does not already exist). So, this implies that the persistency perspective should be created after the creation of the other perspectives, or while creating these other perspectives. If a new perspective is added later, the derivation of persistency features for this new perspective should also be performed. The following steps describe the process of deriving persistency features:

1. **Select a perspective**: The starting point is the *system perspective* (bird's eye view of the product line), as it provides an overview of the key features composing the product line and which are often associated with persistent information.
2. **Inspect the perspective for features that represent or require persistent data**: For each such feature, define a corresponding persistency feature.
   - For example in the QPL's system perspective some features are directly concerned with persistent data such as the *Question* feature that refers to the set of possible question types. Therefore a corresponding *Question* persistency feature will be defined in the persistency perspective.
     Also the *Report Generator* feature manipulates persistent data. The *Report Generator* requires the information about a certain *quiz* taken by a certain *user* to generate a *report*. In this case, we have defined two persistency features to represent this need, i.e. the *Quiz* feature (which represents the information about a specific quiz) and the *User-Quiz Info* (which represents the information about a certain quiz taken by a specific user). Additionally, this analysis also indicates the need for a persistency feature called *User* that represents the information about the user taking a quiz.
3. **Define composition relations relating persistency features.** These composition relations represent whole-part relations between persistency features. They show how features relate to each other from a compositional point of view. Furthermore, these composition relations also allow expressing variability, as a composition relation can be *mandatory* or *optional.*
   - For example in the QPL example, the persistency feature *Quiz* is mandatory composed of the following features: *Question* and *Quiz Element Options* which define respectively the set of possible questions for a quiz and the possible details about the quiz (such as passing score, passing feedback, etc.). Additionally, this *Quiz* feature is optionally composed of a *Question Media* feature that identifies the set of possible media associated with a question. See figure 1(a) for the graphical representation.
4. **Introduce or distinguish between *abstract* and *concrete* persistency features**. Sometimes it may be useful to introduce an *abstract feature* [6] to generalize from a number of more specific features. In that case, the more specific features are called *option features* [6]; they are linked to the abstract feature using a generalization/specification relation. Abstract features must be associated with cardinality rules that govern the maximum and minimum number of option features that should be selected in a valid product configuration.
   - For example in the QPL example, the persistency feature *Question* is defined as an abstract feature that generalizes the features representing the different types of questions. Therefore, the *Question* feature has the following option features: *Sequencing Question*, *True/false Question*, *Matching Question*,

Multiple Choice Question, and *Fill the Blank Question*. We can specify that at least one question type has to be selected and there is no maximum limit in the number of question types used. So the cardinality is expressed as "*1:-*" (see figure 1(a)).

5. **Define the inter-perspective dependencies for the persistency perspective features defined so far**. Inter-dependencies express dependencies between features belonging to one single perspective.
   - For example within the QPL persistency perspective, if we consider the *Question* feature (figure 1.a) and the *Quiz* feature (figure 1.b), we want to express the constraint that for a *Question* feature, selecting the *Assessment Media* feature also requires selecting the *Question Media* feature. This translates into the following inter-perspective dependency: `Assessment Media requires Question Media.`
6. **Define the intra-perspective dependencies**, i.e. dependencies between the features of the perspective selected in step 1 and the features of the persistency perspective.
   - For example in the QPL example, selecting the *Self Assessment* feature (from the system perspective) implies that the data for the questions assessment should also be selected, i.e. the *Question Assessment* persistency feature should also be selected. This translates into the following intra-perspective dependency: `System. Self Assessment requires Persistent. Question Assessment`
7. Repeat steps 1 to 6 to extract persistency features from all existing perspectives (Functional perspective, User's perspective, etc.).

In this way, the persistency perspective is incrementally created.

3.2. Validating the Consistency and Completeness of the Overall Model

Due to the often-tangled relation between data and functionality, a validation for the persistency perspective generated in the previous step is recommended. This is a two-step process motivated by the work in [19] [20], as follows:

1. Validate the consistency of the persistency perspective and its associated inter-dependencies and refine when necessary. This means verifying the following:
   a. Check if no persistency features are unintentionally missing, i.e. the defined persistency features provide an overview of the data concepts required by features of the product line (in all perspectives). This missing of features may be due to errors made during the definition of the persistency perspective or due to missing features in the other perspectives. In this way, this validation is also to some extent a validation of the completeness of the other perspectives.
   a. Check for duplication of persistency features or redundant persistency features, i.e. whether some features have the same semantics (based on their decomposition hierarchy) and actually represent the same persistency feature (this can occur because the same persistency features may originate from different perspectives). In case there is a need for this duplication the "*same*" [21] dependency should be used to treat these redundant features as one.

b. Validate the consistency of the defined inter-dependencies between features of the persistency perspective by checking whether conflicts exist among them (taking into account features related by "same" dependency) yielding to conflicts within the one perspective.

2. Validate the intra-dependencies between the persistency perspective and the other perspectives. This is to some extent a validation of the overall model, it means verifying the following:
   a. The intra-dependencies between features of the different perspectives and the persistency perspective are complete. For this purpose the overall global model needs to be inspected. The global model is created by using the "*same"* dependency as a merge operator that enables merging perspectives based on their common features. This merging of common features allows linking together the different perspectives based on their feature commonality. Furthermore the intra perspective dependencies link features of one perspective to their related features in other perspectives. This linking of related features as well as merge of common features allows gluing together the different perspectives and therefore obtaining a global model.
   b. Within the global model identify and resolve any conflicts found within the intra-perspective dependencies defined.

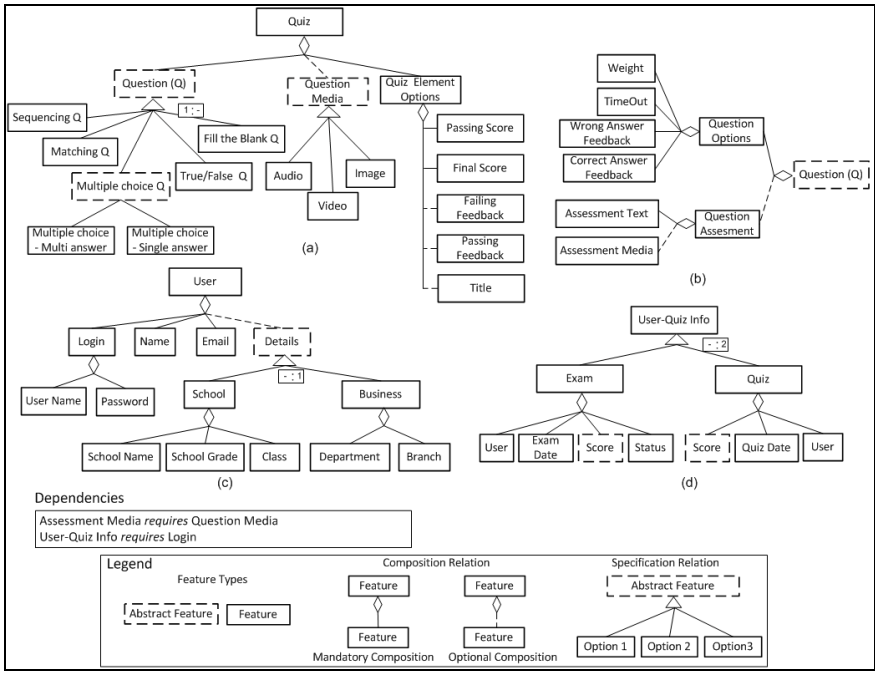


Fig. 1. Persistency perspective for the QPL.

Figure 1[2] shows the result of applying this process to the Quiz product line, Figure 1.a shows the *Quiz* persistency feature, Figure 1.b shows the *Question* persistency feature, Figure 1.c shows the *User* persistency feature (this was mainly derived from features in the User perspective and the Functional perspective), and Figure 1.d shows the *User-Quiz Info* persistency feature which represents information about a specific quiz taken by a specific user. Figure 1 also shows the (inter-perspective) dependencies between the features of this persistency perspective.

- `System.Exam requires Persistent.Termination Page`
- `System.Muliuser requires Persistent.User`
- `Users.School requires Persistent.School`
- `Users.Bussiness requires Persistent.Bussiness`
- `System.Self Assessment requires Persistent.Question Assessment`

Listing 1. Intra-perspective feature to feature dependencies for the persistency perspective.

As already mentioned, it is also important to identify how features affect each other, in terms of intra-perspective feature-to-feature dependencies which tie together the different (related) perspectives to create a global model of the system. The set of intra-perspective feature-to-feature dependencies for the persistency perspective of the quiz product line is given in Listing 1.
Once the persistency perspective is defined and all perspectives are consistent, the next step is to use this information in the database modeling process to obtain a *variable data model*, which can serve for tailoring the different databases required for the different possible product line members (i.e. products). We describe this in the next section.

4. Establishing a Variable Data Model

The persistency perspective is used to steer the database modeling process in producing a variable data model, i.e. a data model that reflects the different variability needs of the product line.
In general, two scenarios exist for defining a data model: a *centralized design* and a *decentralized design* [22]. In centralized database design, the database model is defined in one step, and as a result one global database model is defined. In decentralized database design, a data model is defined for each user view resulting in a number of data model views. In case a global data model is required, it can be derived via a view integration process where the different segments of the database design are combined to create one global model.
How the database model is defined does not affect how the link between the variability model (which is represented by the Feature Assembly Models) and the data

[2] The legend shown in figure 1 is applicable for all subsequent figures showing Feature Assembly Models

model is achieved. In either case, variability information is associated with the data model to instruct how to derive the possible different data models for the different products of the product line. The data model incorporating variability information is called the *variable data model*. There are basically two options for tailoring a variable data model to the needs of each product: the *materialized view* and the *virtual view* [23]. A materialized view means that the required data model elements are actually extracted from the variable data model and are stored physically. This allows creating a tailored database for the final product composed of tables materialized by these views. While with the virtual view, a central database exists, and each individual product has its own view (or sets of views) on this common database.
Modeling a variable data model is a two-step process; first *persistency features* are mapped into *data concepts*. Secondly, the variability of these data concepts is explicitly specified in the defined data model.

4.1. Mapping Persistency Features to Data Concepts

The persistency perspective provides the information about the required persistent data and their variability. Therefore, features in the persistency perspective are mapped to data concepts. To illustrate the mapping, we will use the EER model [24] to represent the data model. However, any other data modeling techniques (such as ORM [25] or UML [26]) can also be used.
All features in the persistency perspective should map to a corresponding data concept. As we are using EER, a data concept can be either an *entity* or an *attribute*. We have defined two annotation to represent this mapping, namely `<<maps_to>>` and `<<relates_to>>`.
`<< maps_to>>` establishes a one-to-one mapping between a feature and a data concept (i.e. entity or attribute).
For example:

- `Persistent.Question <<maps_to>> Data_Model.Question`
- `Persistent.Passing_Score <<maps_to>> Data_Model.Quiz.Passing_Score`

It should be noted that data concepts could be *related* to any feature within any of the existing perspectives. To describe this kind of relationship we use the `<<relates_to>>` notation. `<< relates_to>>` identifies an *association relation* between a feature and a data model concept (i.e. entity or attribute). These association relations are important to establish a link between features in general and the data model.
For example:

- `Functional.Question Category <<relates_to>> Data_Model.Category`

Basically, persistency features map to either *entities* or *attributes* in the data model. Guidelines are:

a. Key features (i.e. features that represent a concrete concept or object) map to entities. For example *Persistent.Question* maps to a *Question* entity in the data model.

b. Features expressing details of key features are in general mapped to attributes rather than entities. For example *Persistent.Passing_Score* maps to the *Passing_Score* attribute of the *Quiz* entity.

We now explain how we can indicate the variability of the data model.

4.2. Representing Variability in Data Models

To specify variability of a data model, it was necessary to extend the model notation. For the EER model, this has been done as follows: we introduced annotations that are used to mark the variability of the concepts, i.e. *entities*, *attributes*, and *relations*. A data concept that originated form a *variable* persistency feature (i.e. optional feature) should be a variable concept in the database model. To denote variability of an entity or attribute, it is annotated with `<<variable>>` and we call it a *variable concept*, i.e. its existence is dependent on the selection of the corresponding feature in the product line. For example, a *Question* feature can be optionally composed of *Question Options* (e.g., time out, weight …etc.). In this case, the *Question* entity is linked to an entity named *Question Options* that is marked as variable, i.e. *Question Options* is marked with the `<<variable>>` annotation to indicate this variability.

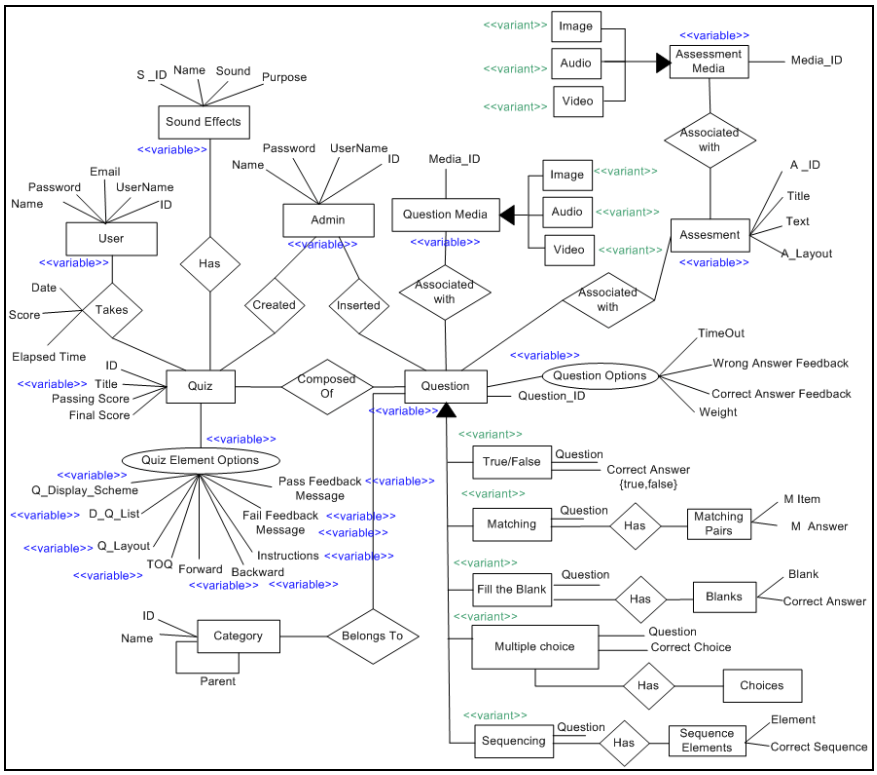


Fig. 2. The variable EER data model for the Quiz product line

A data concept that originates from an *option* persistency feature should also be a variable concept in the data model. To denote variability that is due to a specialization relationship (i.e. option feature), the annotation `<<variant>>` is used. For example in the QPL persistent perspective, there are five different option features of *Question*: *Sequencing, True-False, Matching, Fill the Blank,* and *Multiple Choice*. These *Question* option features are mapped to entities (each feature is represented by an entity) and each entity is annotated with `<<variant>>` to indicate their variability and the fact that they are derived from option features (see figure 2).

Please note that the resulting data model may not be complete. Features defined in the persistency perspective may not completely define all entities, attributes, relationships and constraints of the data model. Rather they only define those concepts related to variability of the product line. Therefore, it is up to the data designer to complete the data model.

Figure 2 shows the variable EER data model for the QPL example. Figure 2 shows that attributes as well as entities can be annotated as variable database concepts. Note that when an attribute is marked as *variable* it should not be used as a *primary key* because it is not guaranteed to exist in all variants of the schema. For the same reason, when using it as a *foreign key* care should be taken as in some schema variants it may not be used. (Note that if no non-variable key is available an artificial key can be introduced.) Also it may be useful to consider putting variable attributes in a separate entity than the primary entity they are attributes of. This will allow easier tailoring of the data model when the variable attributes are not selected. *Quiz Preferences* is an example of such practice.

5. Deriving Tailored Product Data Schemas

As we have explained, a *variable data model* defines the variability of the concepts involved in the variability of a software product line. However, it does not only define the variability of the concepts, it also links the variability of these concepts to the variability of the application features. This enables traceability, i.e. it makes it possible to trace the variability

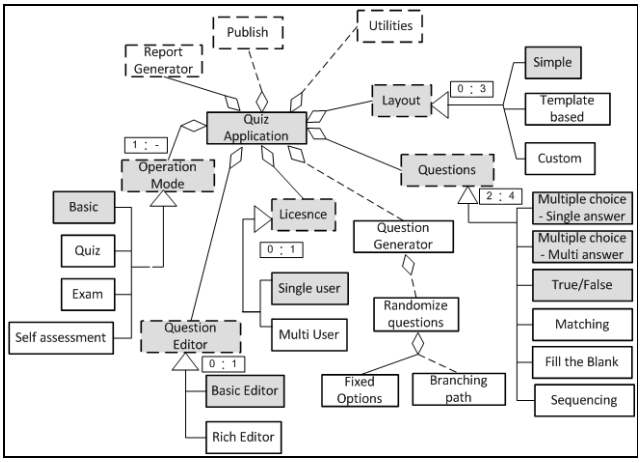


Fig. 3. System perspective of Quiz product 1

from application to database and vice versa. This allows tailoring the data schema of each product to match the data requirements of its features.

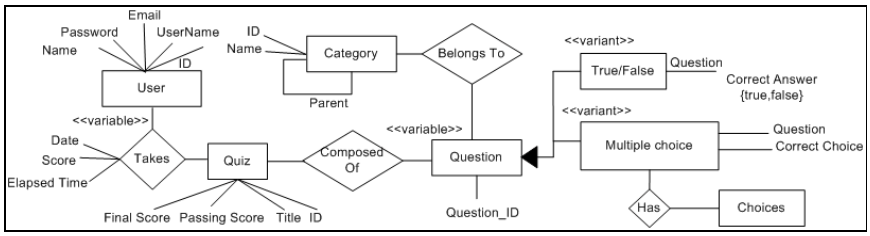


Fig. 4. EER data model for product 1

To demonstrate this, we defined two possible products of the quiz product line, Quiz product 1 is a simple quiz, that contains the following system features *[Quiz, License {Single User}, Layout {Simple}, Questions {True/False, Multiple Choice}, Operation Mode {Basic}, and Question Editor {Basic Editor}],* figure 3 shows in grey shade these selected features . Figure 4 shows the EER data model (view) for Quiz product 1. In listing 2, the relevant mappings are shown, which express the link between the data model and the features.

```
System.Quiz Application <<relates_to>> Data_Model.User
System.Quiz Application <<relates_to>> Data_Model.Quiz
Persistent.Questions  <<maps_to>> Data_Model.Questions
Persistent.True/False  <<maps_to>> Data_Model.True/False
Persistent.Multiple Choice Single Answer <<relates_to>> Data_Model.Multiple Choice
Persistent.Multiple Choice Multi Answer <<relates_to>> Data_Model.Multiple Choice
```

Listing 2. Feature Assembly-to-data model mappings of Quiz product 1

Quiz product 2 shows a more complex example. It is a Quiz and Exam application, and it contains the following system features *[Quiz, License {Multi User}, Layout {Simple, layout, Template Based}, Questions {Sequencing, Matching, Multiple Choice}, Operation Mode {Quiz, Exam}, and Question Editor {Rich text editor}, Report Generator]*. Figure 5 shows the selected features (in grey shade) that

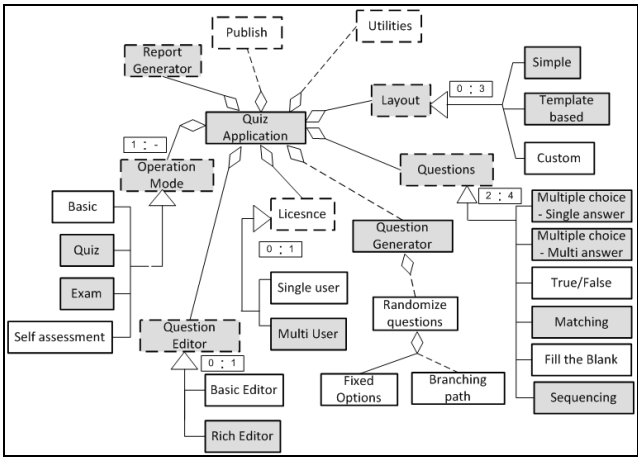
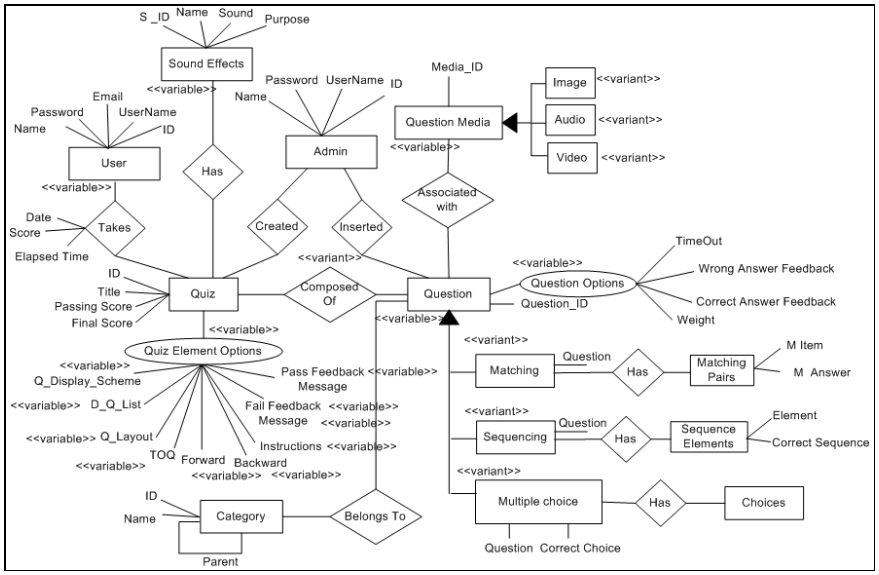


Fig. 5. System perspective of Quiz product 2

represent Quiz product 2. Figure 6 shows the EER data model (view) for Quiz product 2, and listing 3 contains the relevant mappings.



EER data model for product 2

```
System.Quiz Application <<relates_to>> Data_Model.User
System.Quiz Application <<relates_to>> Data_Model.Quiz
Persistent.Questions  <<maps_to>> Data_Model.Questions
Persistent.Multiple Choice Multi Answer <<relates_to>> Data_Model.Multiple Choice
Persistent.Matching <<maps_to>> Data_Model.Matching
Persistent.Sequencing <<maps_to>> Data_Model.Sequencing
System.Multi User <<relates_to>> Data_Model.Admin
System.Multi User <<relates_to>> Data_Model.Question Options
System.Multi User <<relates_to>> Data_Model.Quiz Element Options
Persistent.User <<relates_to>> Data_Model.User
System.Exam <<relates_to>> Data_Model.Sound Effects
System.Multi User <<relates_to>> Data_Model.Question Media
```

Listing 3. Feature Assembly-to-data model mappings of Quiz product 2

6. Conclusion and Future Work

In this paper we have discussed the need for modeling data variability as well as application variability when designing software product lines. We have extended the

Feature Assembly Modeling technique used for feature modeling to also support data variability. For this purpose, we have defined a *persistency perspective* in which features are defined from the point of view of their need for manipulating persistent data in the product line. This persistency perspective is derived from the other perspectives defining the product line. We have also shown how the persistency perspective can be used to define a *variable data model*. A variable data model is a data model annotated with variability information. As such, the actual data model for a product of the product line can be tailored to meet only the requirements of this specific product. This may simplify the process of accessing data, resulting in simpler queries and avoids dummy values in insert and update queries. Another advantage is that it optimizes storage space and allows for different implementation (e.g., for a simple product a full fledged DBMS may not be needed) In addition, the link between the features of the product line and the variable data model is maintained, such that automatic derivation of the actual data model for an individual product is possible.
In this paper, we used and extended EER to express the variable data model, however in a similar way, other data modeling techniques (e.g., UML, ORM) can be used.
Future work includes tool support for the modeling the persistency perspective (this is part of the tool support for the overall feature assembly modeling approach) and for supporting the mapping to the variable data model. We also plan to apply the approach to an industrial case to validate it further.

8. Acknowledgement

This research is sponsored by IRSIB through the VariBru project (www.varibru.be).

9. References

1. Bosch, J.: Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach. Addison-Wesley, Boston (MA), 2000.
2. Bartholdt, J., Oberhauser, R., Rytina, A.: An Approach to Addressing Entity Model Variability within Software Product Lines. In: ICSEA 2008, pp. 465-471, 2008.
3. Bosch, J.: Software Product Families in Nokia. In: SPLC 2005 , pp.2-6, 2005.
4. Clements, P. C., Northrop, L. M.: A Software Product Line Case Study. Technical Report CMU/SEI-2002-TR-038, November 2002
5. Pettersson, U., Jarzabek, S.: Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. In: ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2005.
6. Abo Zaid, L., Kleinermann, F., De Troyer, O.: Feature Assembly: A New Feature Modeling Technique. In: 29th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Vol. 6412/2010, pp. 233-246, 2010.
7. Bolchini, C. , Quintarelli, E., Rossato, R.: Relational Data Tailoring Through View Composition. In: ER 2007, pp. 149-164, 2007.
8. Nyström, D., Tesanovic, A., Nolin, M., Norström, C., Hansson, J.: COMET: A Component-Based Real-Time Database for Automotive Systems. In Proceedings of the

Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04), IEEE Computer Society Press, May 2004.
9. Tesanovic, A., Sheng, K., Hansson, J.: Application-Tailored Database Systems: a Case of Aspects in an Embedded Database. In: Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS'04), IEEE Computer Society, July 2004.
10. Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O. , Saake, G.: FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In: Proceedings of EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM), pages 1–6. ACM Press, March 2008.
11. Rosenmüller, M., Apel, S., Leich, T., Saake, G.: Tailor-made data management for embedded systems: A case study on Berkeley DB. Data & Knowledge Engineering ,Vol 68, Issue 12, pp. 1493-1512, 2009.
12. Siegmund, N., Kästner, C., Rosenmüller, M., Heidenreich, F., Apel, S., Saake, G.: Bridging the Gap between Variability in Client Application and Database Schema. BTW 2009, pp. 297-306, 2009.
13. Kang, K., Cohen, S., Hess, J. Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, 1990.
14. Kang, K., Kim, S., Lee, J. Kim, K. Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, In: J. Annals of Software Engineering. vol. 5, pp. 143-168, 1998.
15. Griss, M., Favaroand, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Fifth International Conference on Software Reuse, pp. 76–85, 1998.
16. Eriksson, M., Börstler, J., Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: Obbink and Pohl (eds), SPLC 2005, LNCS 3714, pp. 33–44, pp. 33- 44, 2005.
17. Czarnecki, K. , Kim, C. H. P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report., In: OOPSLA'05 International Workshop on Software Factories, 2005.
18. Abo Zaid, L., Kleinermann, F., De Troyer, O.: Feature Assembly Modelling: A New Technique for Modelling Variable Software. In: 5th International Conference on Software and Data Technologies Proceedings, Vol. 1, pp: 29 - 35, eds: Cordeiro, J., Virvou, M., Shishkov, B., SciTePress, 2010.
19. Abo Zaid, L., Houben, G-J., De Troyer, O., Kleinermann, F.: An OWL- Based Approach for Integration in Collaborative Feature Modelling. In: 4th Workshop on Semantic Web Enabled Software Engineering - SWESE2008, October 2008
20. Sabetzadeh, M., Nejati, S. Liaskos, S. Easterbrook, S., Chechik, M.: Consistency Checking of Conceptual Models via Model Merging. In: 15th IEEE International Requirements Engineering Conference (RE'07), October 2007.
21. Abo Zaid, L., Kleinermann, F., De Troyer, O.: Applying Semantic Web Technology to Feature Modeling. In: The 24th Annual ACM Symposium on Applied Computing, The Semantic Web and Applications (SWA) Track, March 2009.
22. Connolly, T. M. , Begg, C. E.: Database Systems: A Practical Approach to Design, Implementation and Management. Addison-Wesley, 2009
23. Gupta A., Mumick, I. S.: Materialized views: techniques, implementations, and applications. MIT Press, ISBN: 978-0262571227, 1999
24. Thalheim, B.: Extended Entity-Relationship Model. Encyclopedia of Database Systems 2009: pp.1083-1091, 2009
25. Morgan. T., Information Modeling and Relational Databases, Second Edition, Morgan Kaufmann Publishers, ISBN: 978-0-12-373568-3, 2008.
26. Fowler, M.: UML distilled: a brief guide to the standard object modeling language, Addison-Wesley, ISBN-13: 978-0-321-19368-1, 2004