Incremental Variability Management in Conceptual Data Models of 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*—Software Product Line Engineering is an approach to management of diversity in software families. Although several SPLE approaches exist in the domains of industrial software applications, product lines of data-intensive software systems have gained less attention. We use an incremental, delta-oriented technique to handle variability by specifying changes to be made to a core data model to define the data schemas of the products. We present a new merge-prune operator based on the superimposition of models as well as the structural well-formedness rules specified formally in Alloy. Our method provides a modular way to handle variability in data intensive systems. It is scalable with respect to the number of variation points in the system in contrast to the traditional annotative approaches for variability modeling. We have investigated the applicability of our approach by using it in a real-world case study.**

I. Introduction

Software product line engineering is an approach to manage diversity in families of software products. The approach is based on defining the commonalities and variabilities of the products and building reusable platforms. This is commonly done using feature modeling technique. A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system [1]. A feature can be decomposed into several "mandatory", "optional", or "alternative" sub-features and features can have "or", "requires" and "excludes" relationships [2]. A feature model contains the compact representation of all products of software family in terms of features.

According to the essential role of data in software intensive systems, the variability modeling in software intensive product lines must incorporate the diversity in data and data models too. The importance of the variability management in enterprise information systems is highlighted in [3]. Some studies on data model variability concentrate on variability at the domain model level [4]. This approach is not particularly designed to support the data model variability and it does not directly address the design and implementation of the data model. There are two reasons behind addressing the data model variability directly: first, data model is regarded as an independent artifact in many software intensive systems and the variability in data model is not necessarily identical to that in application. Second, as the database model concentrates on data and relations between the data, and as database design and implementation is an important phase of the software intensive systems development; applying reuse strategies in order to construct a data model for the family is a key factor to manage the variability of the product family.

The presented approach on variability management is based on the delta-oriented programming (DOP) method [5]. In DOP, a core module implements the mandatory features of the product family and delta modules are related to the alternative or optional features. A product is created by applying delta modules to the core module incrementally. As the basic idea of DOP is not restricted to any particular modeling or programming language, we apply it to database modeling to design and implement the product data model. We assume that the core model consists of the mandatory and some selected alternative and optional features and then build a delta model for each non-core feature as a modification on the core model.

For the purpose of this work, the entity-relationship (ER) model has been chosen as a standard method to represent the abstract and conceptual model of database design; as a result, the core and deltas are commonly understood by database designers. A core module (model) is an ER model, which includes entities, relationships and attributes (collectively referred to as *database objects*) of mandatory and a number of selected alternative and optional features. Similar to DOP, the data model of a product is generated automatically by applying the deltas corresponding to the selected (non-core) features of a given product configuration to the core. In the original DOP method, delta modules are described in a textual notation, expressing the transformations on the model. To keep our notations uniform, we use annotated ER diagrams to express delta transformations, in which database objects are annotated by various tags such as *Add*, *Remove*, etc. (see Section III). It is important to note that our method is not classified as an annotative method to variability management in which model elements are annotated by the corresponding features.

An alternative to DOP is feature-oriented approach that a specific feature is implemented in a feature module containing related data model elements. A feature module only replaces or extends parts of the implementation of core, but a delta module can remove parts of the implementation as well as extend or replace the implementation of other deltas [6]. In feature-oriented methods such as [7], [8] the database model is created by composing the feature modules (using superimposition) and they cannot transform (a part of) the model to a new one. In some cases, the implementation of a feature requires some changes in the model that contains changes in element types, for instance, an attribute is changed to an entity. The attribute must be removed and a new entity must be added. The methods

in [7], [8] cannot support such changes. Accordingly, a more complex composition operation is needed, so we present the merge-prune operator for this purpose (Section IV-A). The database objects are annotated to handle the transformation operation. As a result, the presented method supports the mentioned cases as well.

In order to produce a consistent database model from a valid product configuration, inconsistencies between delta models should be recognized prior to the development of the product database model. For instance, suppose that *Delta1* removes entity *A* from the core and *Delta2* adds an attribute to the same entity. Obviously, applying both *Delta1* and *Delta2* in the same product is not possible; hence these two delta modules are not consistent in this case. The consistency rules are checked during the database model generation.

To provide a formal semantics for our method, we translate database meta-model and merge-prune operator into an Alloy specification [9] and exploit the tool support, provided by Alloy Analyzer, to create instances of the meta-model and database as well as check the correctness of the merge-prune operator. Alloy is a lightweight specification language based on relational logic, which is relatively easy to learn and use compared to other specification languages.

Our case study is a family of university software systems. This case study is part of a project which is currently being done in the Software Architecture Laboratory in University of Tehran on developing a middle-sized product line for university information systems aimed at realizing and addressing problems in development of information systems product lines. Our main contributions in this paper are summarized below:

- Providing a modular and scalable variability management method for data models.
- Defining a merge-prune operator on ER models provided with formal semantics (relational logic).
- Applying our method on a real-world case study to demonstrate the applicability of the techniques.

II. Preliminaries

A. Feature Model

A feature model is a representation of the features of a product family. Features are organized into a tree called feature diagram. A parent feature has a number of mandatory, optional and alternative sub-features. Features can have "or", "exclude" ("xor") and "require" relationship. Fig.1 illustrates our running example of a simplified university product line feature model emphasizing the features related to the data model.

A feature model can be represented by a corresponding propositional logic formula in terms of a set of boolean variables [10]. Each boolean variable corresponds to a feature and its value indicates if the feature is included or excluded.

A **feature model** is a pair $(\mathcal{F}, \psi_{\mathcal{F}})$ where $\mathcal{F} = \{f_1, ..., f_m\}$ is the set of features and ψ_F is a propositional logic formula representing the constraints among features. The ultimate formula ψ_F is the conjunction of implications from:

- Each child feature to its parent feature

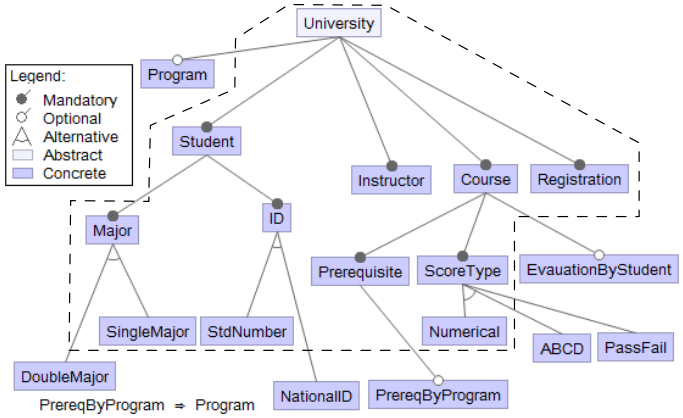

Fig. 1. Simplified university feature model

- Each parent to its mandatory child features
- Each parent to or/xor of its children that have an or/xor relationship
- Each feature f to other features that f requires
- Each feature f to the negation of other features that f excludes

For instance, a part of $\psi_{\mathcal{F}}$ of the simplified university feature model is represented by:
University $\wedge$ *(University* $\rightarrow$ *Student)* $\wedge$ *(Student* $\rightarrow$ *University)* $\wedge$ *(Program* $\rightarrow$ *University)* $\wedge$ *(Student* $\rightarrow$ *Major)* $\wedge$ *(Major* $\rightarrow$ *Student)* $\wedge$ *(ID* $\rightarrow$ *Student)* $\wedge$ *(Student* $\rightarrow$ *ID)* $\wedge$ ...

A set of features f_c satisfies $\psi_{\mathcal{F}}$ ($f_c \models \psi_{\mathcal{F}}$), if $\psi_{\mathcal{F}}$ is $true$ after substituting the variables related to the existing feature(s) with $true$ and non-existing feature(s) in f_c with $false$. By $f_c \models \psi_{\mathcal{F}}$ we mean that f_c corresponds to a valid product configuration.

A set of features $f \subseteq \mathcal{F}$ is consistent, if we can extend it to f_p in a way that $\mathcal{F}_p$ is a valid product configuration. In other word, $\exists \mathcal{F}_p \subseteq \mathcal{F} \cdot (f \subseteq \mathcal{F}_p) \wedge (\mathcal{F}_p \models \mathcal{F})$. In this case, the features in f are consistent with the feature model constraints, and can be regarded as partial configurations.

B. ER Models

"The entity-relationship data model perceives the real world as consisting of basic objects, called entities, and relationships among these objects" [11]. As we represent the core and deltas by entity-relationship models, here we define the elements of an ER model.

"An entity is an object in the real world that is distinguishable from all other objects" [11]. The names of the entities are selected from the *EntityName* domain. Each entity has some describing properties called attributes. We assume *Attribute* to be the set of all attributes of the entities. Each attribute takes its name from the domain of *AttributeName* and its value from *DataType*. The primary key is a set of attributes whose values uniquely determine each instance of entity. An **entity** E is a triple $E = (name,\ A,\ PK)$ where $name \in EntityName$, $A \subseteq Attribute$ is the set of attributes of E and $PK \subseteq A$ is the primary key of E.

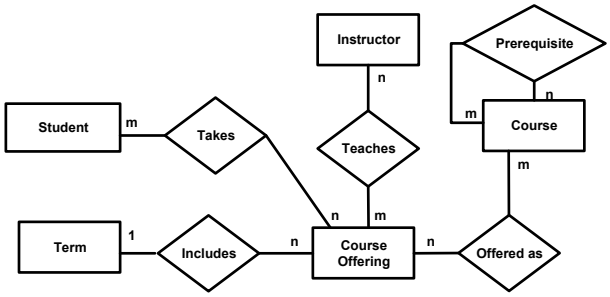

Fig. 2. Core ER Model

In the entity and relationship definitions, A is a set of attributes. An **attribute** A is a tuple *A = (name, T)* where *name* $\in$ *AttributeName* and *T* $\in$ *DataType* where *DataType* is the set of defined data types. To keep it simple, we assume the name of the attributes are unique (or we add the name of the entity as the prefix to the attribute's name).

"A relationship is an association among several entities" [11]. A **relationship** R is a triple $R = (name,\ A, Name_{entity})$ where *name* $\in$ *RelationshipName*, *A* $\subseteq$ *Attribute* is the set of attributes *A* and $Name_{entity} \subseteq$ *EntityName* is the set of entities related to the relationship.

Note that, for simplicity we assume that an ER diagram has only binary relationships. Also, we assume that the sets related to the name of the *entity*, *relationship*, and *attribute* are disjoint.

An ER model describes a database in terms of entities and their relationships. For instance, Fig.2 shows a sample ER model (attributes are not shown in the figures of the paper to keep them simpler). An **entity-relationship** model *ER* is a tuple *(E, R)* where $E \subseteq$ *Entity* and $R \subseteq$ *Relationship*.

Collectively, we refer to all the elements of an ER model as database object. We define *DBO* as the set of all database objects: *DBO = Entity* $\cup$ *Relationship* $\cup$ *Attribute*

As we will see in Section IV, during composition of the deltas and the core, we may have several versions of the same database objects in a model. Hence, we define the following equivalence relation: $\forall dbo_1, dbo_2 \in$ *DBO*, we define $dbo_1 \equiv dbo_2$ if they have the same name.

III. Incremental Variability Modeling Method

In the original delta-oriented programming (DOP), the implementation of a product family is divided into a core module and a set of delta modules [5]. The core module usually implements a complete product for a valid feature configuration. A delta describes a set of modifications to an object-oriented program. One can implement a product by starting from a core module and then applying delta modules to the core in order to include different features. Delta modules can add (remove) code to (from) the product.

We apply delta-oriented approach in the context of database design. The core module contains an ER model including database objects of more common features. Here, we assume that the core model contains at least all mandatory features and a set of selected alternative and optional features (that exist in most products of the family). It is important to note that if such a feature is included in the core, but must be removed in a certain product, there must be a delta defined for removal of this feature. In the sample core (Fig.1, features in dashed line), in addition to mandatory features, the student identity is *StdNumber* and each student studies in only one major (*SingleMajor* feature). Also, only *numerical* score type is supported. The core ER model related to the core features is depicted in Fig.2.

Fig. 3. Annotated ER of DoubleMajor Feature

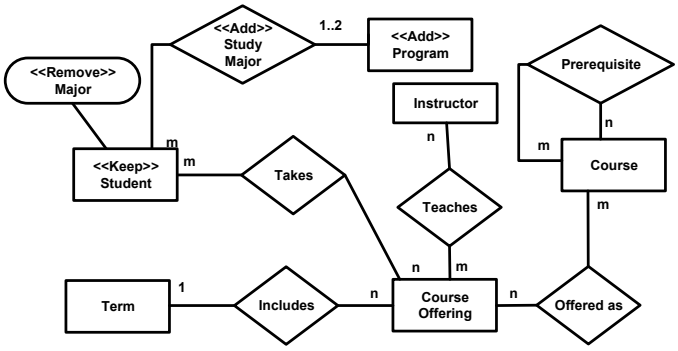

Fig. 4. Merging DoubleMajor to the core

Delta modules specify changes to the core module in order to implement products in the family. In the context of database design, a delta model represents the set of changes to the core model by creating new database objects as well as removing or modifying existing ones. Hence, we present an annotated ER model which represents the changes corresponding to each feature (influencing the data model). The database objects which must be removed from the core are annotated by "Remove" tag and the ones added to the core are annotated by "Add" tag in the delta model. The database objects from the core that are needed by the delta are included in the delta model, annotated by "Keep" tag. For instance, when having the *DoubleMajor* feature, each student can study in two majors at the same time. One way to implement the *DoubleMajor* feature is to remove the *Major* attribute of *Student* and add a relationship between *Major* and *Student* (Fig.3).

In our method, the data model of a product is generated by applying the delta models (related to the selected features of a given product configuration) to the core model sequentially (the order of applying deltas is discussed in Section IV-B). For instance, the result of merging delta related to *DoubleMajor* (Fig.3) to the core is shown in Fig.4.

At the first step, the first delta is added to the core, so the first intermediate model is obtained. Then, at each step, a delta model is applied to the intermediate model generated in the previous step. We extend the database meta-model, by introducing the notions of *Model*, *Core*, *Delta*, and *Tag*.

A database object *dbo* is tagged by the related operation. $\tau(dbo)$ denotes **tag**. $\tau : DBO \rightarrow Tag$ and $Tag = \{Add,\ Remove,\ Keep,\ No\}$. Note that, the core database objects are annotated with the tag "*No*" at the beginning.

A **model** is an annotated ER model corresponding to a valid (partial) feature configuration. Model M is a triple $(DBO_m,\ f_m,\ \tau_m)$ where $DBO_m \subseteq DBO$, is the set of all database objects in M, $f_m \subseteq \mathcal{F}$, f_m is consistent and $\tau_m : DBO_m \rightarrow Tag$.

Core model represents an annotated model related to the mandatory and the selected alternative and optional features. We define $Core = (DBO_{core},\ f_0,\ \tau_0) \in Model$, where $DBO_{core} \subseteq DBO$, $f_0 \subseteq \mathcal{F}$, f_0 is consistent, and $\forall d \in DBO_{core} \cdot \tau_0(d) = No$.

A **delta model** specifies changes to a model (possibly *core*). A *delta* is a triple $Delta = (DBO_d,\ f_d,\ \tau_d) \in Model$, where $DBO_d \subseteq DBO$, $f_d \subseteq \mathcal{F}$, f_d is consistent, and $\forall d \in DBO_d \cdot\ \tau_d(d) \in Tag \setminus \{No\}$.

Note that, in case of changing the entities attached to a relationship, we remove the relationship and add a new relationship in the related delta model. This is because we do not include the "attachments" as first-class objects in our models.

Sometimes, a delta A needs to modify an element E which is not in the core, but is added in another delta B that is required by A. In this case, A also includes E with *Add* tag, containing only the modifications to the version of E in B. As we will see, the merge-prune operator takes care of correct merging of the two versions of E.

For a delta d (corresponding to feature f), let $dep(d)$ denote the set of deltas d_i (corresponding to features f_i) such that f is a child of f_i or f *Requires* f_i. This set denotes the deltas on which d is dependent. We also define *Dep(d)* as the set of all deltas on which d is dependent directly or indirectly: $Dep(d) = TransitiveClosure(dep(d))$.

A. Well-formedness Rules of a Single Delta Model

As each delta must contain a correct ER model, there are some other rules which are extracted from the database metamodel as below:

1) Entity-attribute and Entity-primary key: Each entity has some attributes and primary keys, so the tags of the entity, its attributes and primary keys must be consistent in order to have correct delta models. For example, if an entity in a delta has *Add* tag, its related attributes must have *Add* tag too. As each entity should have at least one attribute defined as the primary key, if we want to remove the primary key of an entity in a delta model, another field must be introduced as the primary key in the same delta model. For a delta = $(DBO_d,\ f_d,\ \tau_d)$, $\forall dbo_d = (n_e, A_e, PK_e) \in DBO_d$, $\forall a \in A_e \cup PK_e$
   - $\tau_d(dbo_d) = Add \Rightarrow \tau_d(a) = Add$
   - $\tau_d(dbo_d) = Remove \Rightarrow \tau_d(a) = Remove$
   - $\tau_d(a) = Add \Rightarrow \tau_d(dbo_d) \in \{Add,\ Keep\}$
   - $\tau_d(a) = Remove \Rightarrow \tau_d(dbo_d) \in \{Remove,\ Keep\}$
   - $\tau_d(a) = Keep \Rightarrow \tau_d(dbo_d) = Keep$
   - $\forall p \in PK_e \wedge \tau_d(dbo_d) = Keep \wedge \tau_d(p) = Remove \Rightarrow \exists p' \in PK_e \cdot\ \tau_d(p') \in \{Add,\ Keep\}$
2) Relationship-attribute: Similar rules exist between a relationship and its attribute(s).
3) Entity-relationship: Each entity is related to some relationships, so the tags of the entity and its relationships must be consistent respectively. It is important that the degree (number of entities that participate in a relationship) of the relationships cannot be changed in the delta models. For a delta = $(DBO_d,\ f_d,\ \tau_d)$, $\forall dbo_e = (n_e,\ A_e, PK_e) \in DBO_d$, $\forall dbo_r = (n_r, A_r, n_{re}) \in DBO_d \cdot n_e \in n_{re}$
   - $\tau_d(dbo_e) = Add \Rightarrow \tau_d(dbo_r) = Add$
   - $\tau_d(dbo_e) = Remove \Rightarrow \tau_d(dbo_r) = Remove$
   - $\tau_d(dbo_r) = Add \Rightarrow \tau_d(dbo_e) \in \{Add,\ Keep\}$
   - $\tau_d(dbo_r) = Remove \Rightarrow \tau_d(dbo_e) \in \{Remove,\ Keep\}$
   - $\tau_d(dbo_r) = Keep \Rightarrow \tau_d(dbo_e) = Keep$

B. Well-formedness Rules Among Several Delta Models

Delta models apply changes to the core model to generate a concrete product model. For a specific valid configuration, the tags of the database objects in one delta model must be checked against the tags of the database objects of other delta models and the core. There are some restrictions for defining the deltas:

1) If a database object in a delta has *Add* tag, the database object must not exist in the core or it must be removed by a dependent delta. For a *delta* = $(DBO_d,\ f_d,\ \tau_d)$ and *Core* = $(DBO_{core}, f_0, \tau_0)$: $(\forall d \in DBO_d \cdot\ \tau_d(d) = Add) \Rightarrow ((\nexists c \in DBO_{core} \cdot\ c \equiv d) \vee (\exists d' = (DBO_{d'}, f_{d'}, \tau_{d'}) \in Dep(d),\ \exists e \in DBO_{d'} \cdot (e \equiv d) \wedge (\tau_{d'}(e) = Remove)))$
2) If a database object in a delta has *Remove* tag, it must exist in the core or added by a dependent delta.
3) If a database object in a delta has *Keep* tag, it must exist in the core and it must not change by all dependent deltas.

It is important to note that an object may be added more than once (same for keep and remove) as long as the above rules are satisfied. This makes the rules independent of the specific order of applying deltas.

IV. Product Derivation

According to the DOP technique, the data model of a product is generated by applying the delta models related to the selected features of a given product configuration to the core model.

A. The Merge-Prune Operator

The proposed merge-prune operator is based on the superimposition technique. "Superimposition is the process of composing software artifacts by merging their corresponding substructures on the basis of nominal and structural similarity" [12]. The superimposition of two ER models merges the models in a way that the final model contains all the elements. Entities or relationships with the same name will have all

attributes and attributes with the same name are replaced in the element. We ignore the variability in attribute data type, because it may lead to data-type inconsistency.

The merge-prune operator $\oplus$ applies a delta d to a model m. Let $d = (DBO_d,\ f_d,\ \tau_d)$ and $m = (DBO_m,\ f_m,\ \tau_m)$, the result of $m \oplus d$ is the model $r = (DBO_r,\ f_r,\ \tau_r)$ where

- $\oplus :\ model \times model \rightarrow model$
- $f_r = f_m \cup f_d$ and f_r is consistent.
- **Merging Entities**: $\forall dbo_m = (n_1,\ A_1,\ PK_1) \in DBO_m, \forall dbo_d = (n_2,\ A_2,\ PK_2) \in DBO_d \cdot dbo_m \equiv dbo_d \Rightarrow \exists (n_3,\ A_3,\ PK_3) \in DBO_r \cdot (n_1 = n_2 = n_3) \wedge (A_3 = A_1 \cup A_2) \wedge (PK_3 = PK_1 \cup PK_2)$
- **Adding Entities**: $\forall dbo_d = (n_2,\ A_2,\ PK_2) \in DBO_d \cdot (\nexists dbo_m = (n_1,\ A_1,\ PK_1) \in DBO_m \cdot dbo_m \equiv dbo_d) \Rightarrow \exists (n_3,\ A_3,\ PK_3) \in DBO_r \cdot (n_3 = n_2) \wedge (A_3 = A_2) \wedge (PK_3 = PK_2)$
- **Keeping Existing Entities**: $\forall dbo_m = (n_1,\ A_1,\ PK_1) \in DBO_m \cdot (\nexists dbo_d = (n_2,\ A_2,\ PK_2) \in DBO_d \cdot dbo_m \equiv dbo_d) \Rightarrow \exists (n_3,\ A_3,\ PK_3) \in DBO_r \cdot (n_3 = n_1) \wedge (A_3 = A_1) \wedge (PK_3 = PK_1)$
- Similar rules exist for merging, adding and keeping relationships.

The resulting tags are defined according to the rule that says the tag of an object in the resulting model is always equal to the tag of the same object in the delta. For those objects in m that are not affected by the delta, we keep their tags.

- $\forall dbo_d \in DBO_d \Rightarrow \exists dbo_r \in DBO_r \cdot dbo_r \equiv dbo_d\ and\ \tau_r(dbo_r) = \tau_d(dbo_d)$
- $\forall dbo_m \in DBO_m . (\nexists dbo_d \in DBO_d \cdot dbo_m \equiv dbo_d) \Rightarrow \exists dbo_r \in DBO_r \cdot dbo_r \equiv dbo_m\ and\ \tau_r(dbo_r) = \tau_d(dbo_m)$

For instance, the result of applying merge-prune operator to the core model and the delta model of the *DoubleMajor* feature (Fig. 3) is shown in Fig. 4.

B. Order of Applying Deltas

To find the sequence of applying deltas, we create a dependency graph for the selected features and then use the topological sort algorithm to find a proper sequence of applying the related deltas to the core. Then, the merge-prune operator applies the deltas to the core in that order. The dependency graph is created as follows:

1) When feature *A* is not a part of the core but is selected in the product configuration, there is a node *A* in the dependency graph.
2) When feature *A* is part of the core and is *Not* selected in the product configuration, there must be a node A' in the dependency graph, representing the delta defined for removal *A* from the core.
3) When feature *A* is the parent of feature *B* and *A* is not in the core (so, neither *B*), there is an edge from node *A* to node *B*, meaning that *A* must be applied before *B*.
4) When feature *B* requires a feature *A* which is not in the core, there is an edge from node *A* to node *B*.
5) When feature *D* has 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*.

After applying related deltas to the core, all the database objects with *Remove* tag are removed from the model in the post-process. This process removes attributes of entities and relationships that have *Remove* tag, too.

C. Consistency of the Resulting Data Model

In all intermediate models during application of deltas for a configuration, we have some properties as described below. These properties can be easily proved based on the well-formedness rules and the merge-prune semantics (Section V).

- If a database object has *Add* tag in the model, it is not in the core model or it must have *Remove* tag in one of the previous intermediate models.
- If a database object has *Remove* tag in the model, it must be in the core or it must have *Add* tag in one of the previous intermediate models.
- If a database object has *Keep* tag in the model, it must be in the core and has the same tag in all the previous intermediate models.

The above properties can be stated formally. For instance, the first one can be stated as: Let $m_0, ..., m_n$ be any sequence of the models obtained from applying topological sort to the dependency graph, when $m_0 = (DBO_0, f_0, \tau_0)$ is the core, $m_1, .., m_{n-1}$ are intermediate models ($m_i = (DBO_i, f_i, \tau_i)$) and $m_n = (DBO_n, f_n, \tau_n)$ is the final data model:

$(\forall dbo_r \in DBO_n \cdot\ \tau_n(dbo_r) = Add) \Rightarrow ((\nexists dbo_0 \in DBO_0 \cdot dbo_0 \equiv dbo_r) \vee (\exists i,\ 0 < i < n, \exists dbo_{mi} \in DBO_{mi} \cdot\ dbo_r \equiv dbo_i \Rightarrow\ \tau_i(dbo_{mi}) = Remove))$

V. Analyzing in Alloy

We translate extended database meta-model (Section III), two sets of well-formedness rules (Section III-A and Section III-B) and merge-prune operator (Section IV) into Alloy specification to formally encode the semantics of our method. Also, we use Alloy Analyzer to confirm the correctness of the method and resulting data model based on the well-formedness rules of deltas and defined merge-prune operator.

A. A Quick Introduction to Alloy

Alloy [9], which is based on first order relational logic, is a formal specification language suitable to identify the right software abstractions at the early stages of software development. Alloy Analyzer, is a constraint solver that takes the specification and the constraints of a model and finds model instances that satisfy them. In spite of the incompleteness of the Alloy analysis, the "small scope hypothesis" [13] (examining all small cases makes finding a counterexample highly probable) strongly encourages the work on limiting the scope (size of the model) [9].

An Alloy model specification is a structured specification. A *signature* plays the role of a type; it defines the set of

```
1   sig Entity { name: one Name, attr: some Attribute, pk: some PK,
      tag: one Tag }
2   sig Model { entity: some Entity, rel: some Relationship, dref: one Delta }
3   sig Delta { entity: some Entity, rel: some Relationship , dep: set Delta }
4   fact { all disj m, m': Model |m.dref != m'.dref }
5   fact coreModelEntity { MO/first.entity.tag in No and
      MO/first.entity.attr.tag in No and MO/first.entity.pk.tag in No }
6   fact coreModelRelationship {
      MO/first.rel.tag in No and MO/first.rel.attrel.tag in No }
7   fact operationAddEntity {
      all d: Delta, disj e: d.entity |e.tag in Add implies e.attr.tag in Add }
8   fact operationKeepEntity {
      all disj d,d':Delta, disj e:d.entity, e':d'.entity |
      e.tag in Keep and e.name = e'.name implies (e'.tag = Keep)
      and (e.name in MO/first.entity.name)}
9   pred MergeModel [ m:Model, m': Delta,m": Model] {
      m".entity.name = m.entity.name + m'.entity.name
      m".entity.attr.name = m.entity.attr.name + m'.entity.attr.name
      m".entity.pk.name = m.entity.pk.name + m'.entity.pk.name
      all e:m.entity, e': m'.entity, e":m".entity |
        e".name = e'.name and e.name = e'.name implies
        e".tag = e'.tag and MergeAttributePK[e,e',e"]
      all e: m.entity, e': m'.entity, e1:m".entity |
        e.name != e'.name and e1.name = e.name implies
        e1.tag = e.tag and KeepAttributePK[e,e1]
      all e: m.entity, e': m'.entity, e1:m".entity |
        e1.name = e'.name and e.name != e'.name implies
        e1.tag = e'.tag and AddAttributePK[e,e1] ... }
10  fact mergeRef { all m: Model |m.dref.DO/next = m.MO/next.dref}
11  fact merge { all m:Model-MO/first |
        MergeModel [m.MO/prev, m.MO/prev.dref,m]}

12  assert ConsistencyRules {
13    all m: Model - MO/first, e:m.entity, n:e.name |e.tag in Keep and
        n.~name in MO/first.entity implies n.~name.tag in No
14    all m: Model - MO/first, e:m.entity, n:e.name |e.tag in Keep implies
        n.~name.tag in Keep + No and e.name in MO/first.entity.name
15    all m: Model - MO/first, e:m.entity, a:e.attr |
        (e.tag in Remove implies a.tag in Remove)
16    all m: Model, r:m.rel |r.tag in Add implies r.entity.tag in Keep + Add ... }
```

Fig. 5. Some parts of the Alloy specification of the method

elements and possibly the relationship with other elements. *Signature extension* supports classification hierarchy and inheritance. *Facts* are constraints on signatures that always hold. Therefore, they have to be satisfied by all the instances of the model. *Predicates* define reusable constraints which represent an operation. *Predicates* are checked when invoked and the result can be evaluated to true or false [9].

B. Semantic Description in Alloy

In this section, first we convert the extended database meta-model into Alloy specification. Then, we encode the semantics of our method into facts described over the defined signatures. Here, we present some part of the Alloy specification [1]. All elements in the meta-model are mapped to Alloy signatures. For instance, line 1 of Fig. 5 shows Alloy signature for *Entity* indicating that each entity has a name and a tag plus some attributes (at least one) and some primary keys.

As the data model of a product is generated by applying delta models to the core model, at each phase there is a model on which the next delta is applied to. Hence, each delta has a reference to the model which should be applied to it (Fig. 6). The Alloy specification of *Model* and *Delta* is depicted in lines 2 and 3 of Fig. 5. Furthermore, a number of facts are defined in order to describe meta-model constraints in Alloy.

[1]The complete specification is accessible via http://khorshid.ut.ac.ir/∼n.khedri/JCaseStudy/dodd.als

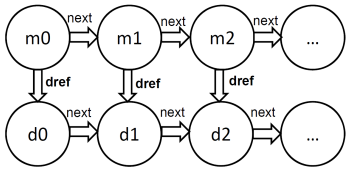

Fig. 6. Model and Delta relation

For instance, the facts in lines 5 and 6 of Fig. 5 show that the database objects in the core must have *No* tag. Here, *MO/first* represents the core model.

We transform the well-formedness rules of the deltas (Section III-A) to Alloy facts. For instance, Fig. 5 line 7 shows that if an entity in a delta model has *Add* tag, all attributes of the entity (*e.attr*) must have *Add* tag.

Further, we translate the well-formedness rules among delta models (Section III-B) into Alloy specification and check their correctness using Alloy simulation. For instance, Fig. 5 line 8 shows that if an entity in a delta model has *Keep* tag, it must be in the core and it must have *Keep* tag in all dependent deltas.

A part of Alloy specification of the merge-prune operator (related to entities and attributes) is depicted in line 9 of Fig. 5. Merge-prune operator merges a model m and its related delta m' and the result is model m''. The "+" operator in Alloy stands for the union operation.

Based on the presented meta-model, each model has a reference to the delta which should be applied to it (Fig. 6); as a result the merge-prune operator is applied to the model and its related delta. The Alloy specification for this constraint is shown in Fig. 5, lines 10 and 11. The first fact means the next delta for a model's delta is the same as the next model's delta. The second fact denotes that the merge-prune operator is applied on the previous model and the previous delta except for the first model (which is the core model). We use the Alloy *Ordering Package* to implement these facts.

C. Analysis

We use Alloy Analyzer to examine the database meta-model constraints and visualize instances of the database meta-model. Here, we show that having the set of rules for the core model and delta models as well as defining merge-prune operator leads us to a correct database model. From the formal specification of our method, Alloy Analyzer automatically generates all possible instances within given bounds and validates the assertions. The database model is checked according to the model consistency rules (Section IV-C). For instance, lines 12, 13 and 14 of Fig. 5 show that if an entity has *Keep* tag, it must have *No* tag in the core and *Keep* tag in other models.

Additionally, each data model is a correct annotated ER model; as a result, we check the database model rules similar to the delta model rules discussed in Section III-A. For example, Fig. 5, Line 15 illustrates that if an entity has *Remove* tag, the related attributes must have *Remove* tag. Also, if a relationship has *Add* tag, the relating entities must have *Add* or *Keep* tags (Fig. 5, Line 16).

No counterexample found after checking the assertions within various given scopes.

VI. 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 [14]. A group of software engineering students have studied six university software systems in our country. One of the systems has four different versions deployed in four 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 four 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 common to all products were considered mandatory. 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 [15]. 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 core and delta module.

We have selected 85 out of 125 features in the whole family as the case study [2]. The core contains 21 mandatory and three selected optional and alternative features. We build the core in only one annotated ER diagram [3]. There are 61 optional and alternative features which are added to the core by applying a single delta per feature. All deltas are well-formed and, 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, we chose eight products which especially include *Alternative* features (not in core) and *Require* relations.

Note that, our method can handle the independent optional features that their implementations are dependent (known as optional feature problem), if their sets of dependent features are not disjoint.

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

VII. Related Work

In this paper, we propose a delta-oriented method on database design context to build the database model of a product in a family incrementally by starting from a core data model and applying delta models to it in order to implement different features. An approach including the definition of mapping features to model elements also containing feature relations is presented in [16]. The work is similar to our method in maintaining the relation between features and model elements; however our method focuses on representing data model variability and each delta model implements a particular feature.

A. Variability in Database Models

The ADMV process (Addressing the Data Model Variability) [4] is a UML-based approach for SPL data modeling and data integration. ADMV provides a unified and systematic methodology to support a consistent view by using adapters and views to capture data variability in data models. It is important to note that [4] does not consider the logical and conceptual modeling of the database and apply the suggested technique to a UML-based domain model.

Also, [7] focuses on variable schemas in software product lines. The related schemas are selected and superimposed to compose the variable schema. In [8], the work is extended to generate a tailored database schema automatically using features. The similarity between [7], [8] and our method is in building the schema based on superimposition of the ER diagrams. In composing the database schema, there are cases that a database element must be removed and another element added instead. The approaches like [7] and [8] generate the database by composing the schema parts and they cannot transform a model to a new one. As the presented transformational supports the mentioned cases as well.

Moreover, an approach for modeling data variability is provided in [17] which is a part of the overall software product line modeling approach. The authors introduce a method to represent and model database variability in a single model. Note that in [17] the set of variability entities includes entities and attributes; but in our method relationships, relationship attributes and primary keys can be variable too. 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.

In our previous work [18], we presented an approach to manage variability in database design of information system product lines based on DOP technique [5]. The core consists of the DDL scripts of the mandatory and some selected alternative and optional features. Also, each delta consists of the DDL scripts of the changes to the core to implement the features outside the core. The DDL scripts include the related tables, columns, primary keys, foreign keys and constraints to implement the feature in data model. The consistency of the resulting database schema is checked by the database management system (DBMS) when creating the database. In our current approach, on the other hand, the consistency checking is done on the ER models at the design time, independent of the specific DBMS to be used. The textual syntax of the DDL scripts makes defining and managing deltas easier, but when the size of the product family increased, managing the deltas and resolving the conflicts will be complicated. As a result, the

[2]The complete case study feature model is accessible via http://khorshid.ut.ac.ir/~n.khedri/JCaseStudy/CompleteCaseStudy.pdf

[3]The complete annotated ER diagrams are accessible via http://khorshid.ut.ac.ir/~n.khedri/JCaseStudy/UniversityEducationFM.png

presented method in this paper is more understandable with regards to the size of the family. Also, the method of [18] is described solely in SQL language and lacks a mathematical foundation. Here, we have used relational logic as a basis to precisely describe our delta-oriented approach.

B. Model Merging

In [19], an integrated approach for building products in SPL based on merging UML diagrams is proposes. Merging models is a well known method for product derivation process.

The superimposition technique for UML models is used in [12] to provide a tool for UML model composition with variability support. "Superimposition is a technique to merge code pieces belonging to different features" [12]. A framework called "FeatureHouse" is proposed to compose software artifacts in different languages [20]. They apply their approach on UML class diagrams, state diagrams and sequence diagrams to model variability. It is important to note that [12], [19], [20] concentrate on managing variability in general, and not in data model. Instead, our method specifically focuses on database models. Hence, we have made use of the ER meta-model along with its semantics to provide a composition operation which is semantically richer than a general merge operator. For example, our consistency checking rules, make sure that the resulting data model obeys ER well-formedness rules.

An approach is proposed in [21] on relating variability model and design model using UML package merge mechanism. In our work, we allow a delta to remove elements from the core as well as merge elements.

VIII. Conclusion and Future Work

In this paper, we tackled the variability management problem in data models at conceptual database design level. We expounded a new variability modeling technique based on the delta-oriented programming (DOP) technique and superimposition of ER models. Data model of a product is generated by applying delta models to the core model sequentially. We presented a new merge operator, named merge-prune, to apply deltas to the core. The sequence of merging deltas to the core is derived from applying topological sort to the dependency graph of the selected product configuration.

We presented a meta-model for relational database based on DOP and set of rules to have well-formed core and deltas. Also, we checked the consistency of the product data model based on analyzing the tags of database objects. Accordingly, the models obtained from applying deltas incrementally all have correct tags for their elements. We transformed the database meta-model and the merge-prune operator in Alloy specification and analyzed the consistency of the resulting model by Alloy Analyzer through transforming consistency rules to a set of assertions in Alloy.

In comparison to the approach that presents variability in data model [17], our approach supports software product line evolution, because it is modular and can handle the changes related to the feature model well. As we model the variabilities in separate delta models, our method is scalable in practice in comparison with methods like [17] that try to represent all variabilities in a single model. Future work includes adding tool support for the presented method.

References

[1] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again." in *Proceedings of the 11th International Software Product Line Conference*. IEEE Computer Society, 2007, pp. 23–34.

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

[3] N. Kozuka and Y. Ishida, "Building a product line architecture for variant-rich enterprise applications using a data-oriented approach," in *SPLC Workshops*. ACM, 2011, pp. 14:1–14:6.

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

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

[6] S. Schulze, O. Richers, and I. Schaefer, "Refactoring delta-oriented software product lines," in *AOSD*. ACM, 2013, pp. 73–84.

[7] N. Siegmund, C. Kästner, M. Rosenmüller, F. Heidenreich, S. Apel, and G. Saake, "Bridging the gap between variability in client application and database schema," in *BTW*, 2009, pp. 297–306.

[8] M. Schäler, T. Leich, M. Rosenmüller, and G. Saake, "Building information system variants with tailored database schemas using features," in *CAiSE*, 2012, pp. 597–612.

[9] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[10] D. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of the 9th international conference on Software product lines*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 7–20.

[11] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 4th ed. McGraw-Hill, Inc., 2001.

[12] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*. Springer-Verlag, 2009, pp. 4–19.

[13] A. Andoni, D. Daniliuc, and S. Khurshid, "Evaluating the small scope hypothesis," Massachusetts Institute of Technology - Software Design Group, Tech. Rep., 2002.

[14] 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, april 2009.

[15] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[16] F. Heidenreich and C. Wende, "Bridging the gap between features and models," in *2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'07)*, 2007.

[17] L. A. Zaid and O. D. Troyer, "Towards modeling data variability in software product lines," in *BMMDS/EMMSAD*, 2011, pp. 453–467.

[18] N. Khedri and R. Khsoravi, "Handling database schema variability in software product lines," in *The 20th Asia-Pacific Software Engineering Conference (APSEC 2013)*, 2013.

[19] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jezequel, "Reconciling automation and flexibility in product derivation," in *Proceedings of the 2008 12th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 339–348.

[20] S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231.

[21] 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, 2010.