A Framework for Synchronization Between Feature Configurations and Use Cases Based on Bidirectional Programming

Weize Zhao
Peking University
Beijing, China
Email: zhaoweize@pku.edu.cn

Haiyan Zhao
Peking University
Beijing, China
Email: zhhy.sei@pku.edu.cn

Zhenjiang Hu
National Institute of Informatics
Tokyo, Japan
Email: hu@nii.ac.jp

***Abstract*—Model-Driven Development (MDD) is a widely adopted approach to Requirement Engineering (RE). One basic research issue in Model-Driven Requirement Engineering (MoDRE) is requirements validation, which focuses on how to validate whether the requirements models meet stakeholders' needs or not. Several lines of work have been performed on the transformation between feature configurations, which are responsible for specifying a software in feature-oriented approach, and use cases, which are easy to understand and often used to describe system behaviors to stakeholders. However, most of the existing automated derivation methods about feature configurations and use cases are either in one direction or the other. Therefore, after validating the use cases, the adjustment made by stakeholders cannot be traced back to feature configurations automatically. In this paper, we focus on synchronization between these two vital software artifacts. And we propose a framework that uses putback-based bidirectional programming to guarantee the correctness of this synchronization.**

***Index Terms*—Feature model, Use case, Synchronization, Bidirectional Programming.**

I. Introduction

Requirement validation has been one of the important issues in requirement engineering, especially in systematic software reuse. Requirement models of software reuse approaches are usually highly abstract and difficult for untrained users to read and understand. As an instance, Feature-Oriented Domain Analysis (FODA) method [1] is a widely adopted software reuse approach in business and technical domains which uses a feature model to capture commonalities and variabilities of a domain, and a feature configuration to describe a specific product in the domain. Although feature-oriented approach has been greatly developed since it was orignally proposed, feature configurations are still too abstract for untrained users to understand.

In the other hand, use cases model the functionality of systems at a similar level of abstraction but from a user perspective. They are also widely adopted as requirement models, especially to communicate with stakeholders.

To achieve the advantages of both sides, several lines of work have been performed on the transformation between feature configurations and use cases. But the transformation process proposed in these works are mostly unidirectional.

For example, Griss et al. [2] provide high-level guidelines to derive feature models from use cases. The transformation information, which is obtained as a by-product of the derivation, is not formally specified. It is hard to apply the information for bidirectional transformation between feature configurations and use cases. In Bonifacio et al.'s work [3], they propose an approach to building transformation information between features and use case fragments. Applying the transformation information, use cases can be derived from a feature model configuration, whose main drawback is that the transformation is unidirectional. The feature configuration can not be updated with a modified use case in this framework. Cazrnecki and Antkiewicz's [4] propose a general template-based approach for mapping feature model configuration to other kinds of models, including use cases. The main limitation of this work is that the process of derivation is not specified explicitly, and the derived model can not be transformed backward to the feature configuration.

In our previous work, we propose a transformation description language (TDL, Section III-A) to specifying the transformation information from feature models to use cases [5]. Rules written in TDL can be applied to derive use cases from feature configurations. However, there is still no corresponding part to derive feature configurations from use cases, which means if a stakeholder modifies the derived use cases, the corresponding feature configuration has to be updated manually. Moreover, transformation information written in TDL implies variability inside use cases. One feature configuration can correspond to variant use cases depending on different excution order of the transformation information, and our previous work just applied the transformation in the default order without considering the variability. There are also some approaches that allows modeling vairability directly in use case. Hajri et al. [6] [7] propose a product line methodology for documenting variability in use cases. However it can't describe the variability within a use case.

In this paper, we propose a framework focusing on the synchronization between feature configurations and use cases based on TDL. Through our framework, a feature configuration can be automatically updated with an adjusted use case,

978-1-5090-3694-3/16 $31.00 © 2016 IEEE
DOI 10.1109/REW.2016.45

IEEE computer society

RE 2016 Workshops, Beijing, China
MoDRE Workshop Paper

and vise versa. The synchronization is not a trivial task to manually carry out, especially considering the following three factors:

- Transformation information written in TDL is unidirectional;
- Variability also exists inside the use cases (as activities in this paper);
- The variability of activities is order-sensitive.

To ensure the consistency between feature configurations and use cases, we synchronize them by using bidirectional transformation (BX, Section III-B) [8] techniques. A BX consists of a pair of functions: a forward one, which generates a view from a source, and a backward one, which takes the original source and an updated view as input, and outputs an updated source where the view has been embedded in. We have implemented the BXs with BIGUL [9], a putback-based bidirectional programming language. With BIGUL, we describe only the behavior of put, and the pair of get function and put function will be generated by the BIGUL compiler. The well-behavedness of the pair of functions is guarenteed by BIGUL.

The remainder of this paper is organized as follows: Section II provides a practical example of the problems our framework addressed. Section III gives some preliminaries on TDL and BXs. Our framework is presented in Section IV. In Section V, we examine threats to the validity. After discussing related work in Section VI, we conclude the work in Section VII.

II. A RUNNING EXAMPLE

Let's consider such a scenario: A stakeholder, who owns a pizza store, finds that ordering pizzas by phone is an inefficient way, especially at a peak time. Inspired by the boom of O2O (Online to Offline) business model, he plans to build an online store so that customers can order the pizzas through the internet.

The project is contracted to a software company that adopts feature-oriented approach to achieve software reuse. In the approach, the characteristics of online stores are denotes by *features*, and a *feature model* is built to organize all the mandatory features and optional features of online stores. Fig. 1 shows a fragment of the feature model. The features in the fragment are related with the function *check out*.

After meeting and discussing with the stakeholder, a requirement analyst generates a configuration of the feature model. Fig. 2 gives the configuration, where only the features in Fig. 1 is explicitly presented.

Given the feature configuration, the stakeholder complains that the configuration is too abstract for him to understand precisely. It's hard to validate whether the captured requirement meets his need or not. Therefore, an automated use case derivation approach based on TDL (Section III-A) is used to generate the corresponding use cases for validating the captured requirement with the stakeholder. Fig. 3a shows one of the automated derived use cases which describes the scenario of checking out.

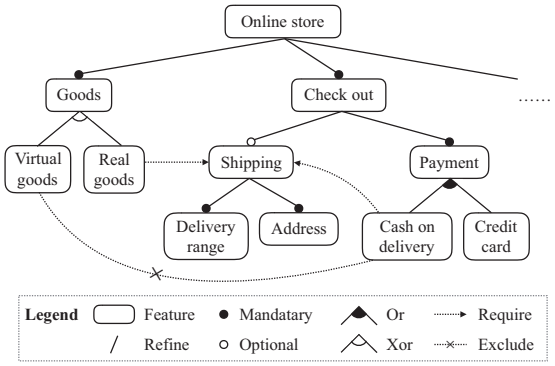

Fig. 1: A fragment of the feature model "online store"

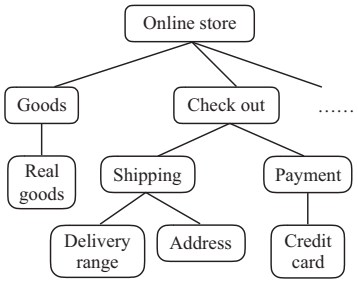

Fig. 2: A fragment of the feature configuration of online store

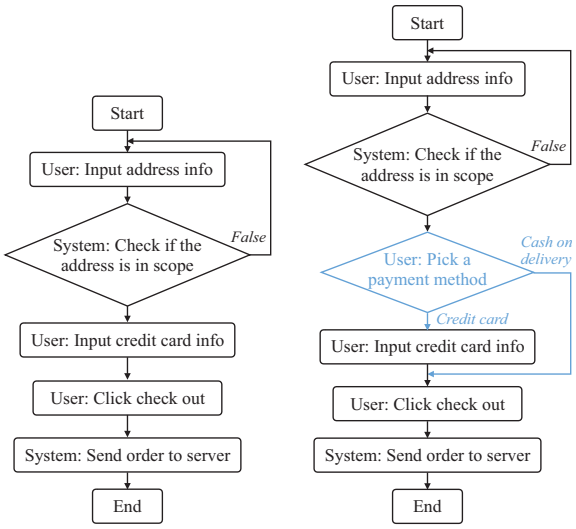

Fig. 3: use case *UC_check_out*

After validating the use cases, the stakeholder is unsatisfied with the process of checking out and want to modify the use case. He thinks that although credit card has become a popular payment method, plenty of customers still prefer cash on delivery. In the other words, he'd like to change the requirement.

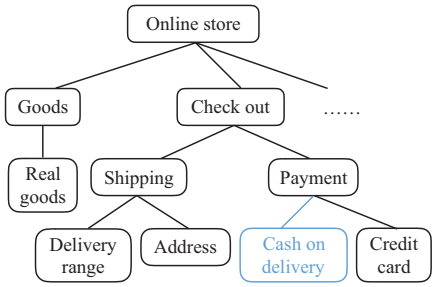

Fig. 4: A fragment of the updated feature configuration

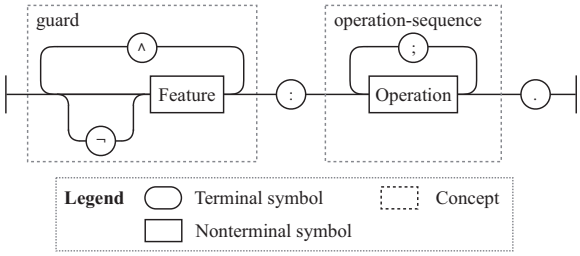

Fig. 5: Core syntax of TDL

The feature model that given in Fig. 1 seems so simple that the stakholder can easily understand the model and adjust the feature configuration directly, it's only a small part of the whole feature model. In fact, even a small scale feature model in practise is usually contains more than one hundred features and dozens of constraints. On the other side, the requirements of adjustment will also be more complicated than just add a "Cash on delivery" in most cases. Therefore, for a stakeholder, adjusting the use case directly is a more feasible solution.

In this case, the use case in Fig. 3a is modified into Fig. 3b. A new decision node *User: Pick a payment method* is inserted, and activity *User: Input credit card info* becomes one of its branch. Another branch which denotes *Cash on delivery* is also added, but with no activities on this branch.

As the core artifact of feature-oriented software product line method, the feature configuration should be updated to satisfy the adjustment of the use case. However, as far as we know, rare work has been done to automatically achieve this goal. We have to manually add features which correspond to the new activities, remove features which correspond to the deleted activities, and apply further adjustment to keep the validity of the feature configuration. In this case, we add feature *Cash on delivery* into the original configuraion. The updated feature configuration is showed in Fig. 4.

It is not a trivial work to manually update a feature configuration with adjusted use cases, even for a simple instance like the above one. Considering not only the consistency between the configuration and the use case should be hold, but also all the constraints of the feature model need to be satisfied, the difficulty and comlexity will be greatly increased when dealing a larger-scale domain.

III. Preliminary

A. Transformation description language

The transformation description language (TDL) is designed for precisely specifing the transformation information from a feature model to a set of use cases [5]. The feature model adopted in TDL, which is proposed in FeatuRSEB [2], consists of two categories of elements, i.e. *features* and *relations*. A *feature* is a user-visible capability of a software system, denoting a cohesive set of individual requirements[10]. By selecting a set of features in a feature model, we can get a feature configuration that describes the capability of a specific software product.

Relations fall into two kinds: *refinement* and *constraint*. *Refinement* relations integrate the features into a tree structure. *Constraint* relations are further classified into *require*, *exclude*, *OR* and *XOR*. In the feature model showed in Fig. 1, as an instance, *Real goods* requires *Shipping*. *Virtual goods* excludes with *Cash on delivery*. *Cash on delicery* and *Credit card* share an *OR* relation, which means if their parent feature *Payment* is selected, at least one of them should be selected as well. *Virtual goods* and *Real goods* are in an *XOR* relation , which means if *Goods* is selected, one and only one of them has to be selected.

To be valid, a feature configuration has to satisfy all the constraints of the feature model, including the ones explicitly indicated in *constraint relations* and those implicitly implied in *refinement relations*.

A use case specifies a set of behaviors performed by the software, which yields an observable result that is of value for stakeholders. It can be denoted in various forms: natural language, pseudocode, activity diagram, etc. [11]. Since we focus on the sequence of interactions inside a use case in this paper, and considering the needs of formal specification, we use structured activity diagrams. And the activities can be further classified into three categories: $Single$, $Branch$ and $Loop$, where both $Branch$ and $Loop$ are compound activities.

TDL formally specifies the transformation information from a feature model to use cases into a set of transformation rules (*X-rules*). According to the core syntax of TDL showed in Fig. 5, each *X-rule* consists of a guard and a sequence of operation. The guard is a conjunction of features' binding states: a and $\neg a$ respectively indicate feature a is selected or removed. The guard specifies the kinds of feature configuration in which the *X-rule* will be active. The operation sequence of an active *X-rule* will be excuted.

TDL supports two categories of operations: *create* and *insert*. Use cases and activities can be created with *create* operations, and *insert* operations is used to insert activities into use cases at the right position relatively. Both the *create* operation and the *insert* operation of each activity will only appear once in the whole set of *X-rules*. TABLE Ib represents the *X-rules* related with *UC_check_out*. The *create operations* of activities are omitted and the detailed attributes of activities are extracted and listed in TABLE Ia.

TABLE I: The X-rules of *UC_check_out*

(a) activity list

Id	Type	Actor	Behavior
A1	Single	User	Click check out
A2	Single	System	Send order to server
A3	Single	User	Input address info
A4	Single	User	Input credit card info
B1	Branch	User	Pick a payment method
C1	Condition		*False*
C2	Condition		*Cash on delivery*
C3	Condition		*Credit card*
L1	Loop	System	Check if the address is in scope

(b) rule list

No.	Guard	Operations (... in *UC_check_out*)
1	*Check out*	create use case *UC_check_out* with activity *Start*, *End*; insert *A1* before *End*; insert *A2* after *A1*;
2	*Shipping*	insert *L1* with *C1* before *A1*; insert *A3* after *C1*;
3	*Payment*	insert *B1* before *A1*;
4	*Cash on delivery*	insert *C2* as a condition of *B1*;
5	*Credit card*	insert *C3* as a condition of *B1*; insert *A4* after *C3*;

For a detailed description of the TDL's syntax, use and case studies please refer to [5].

B. Bidirectional transformation

Bidirectional transformations (BX) provide a novel mechanism for synchronizing the contents of two related pieces of data, one as *source* and the other as *view* [12]. One of the BX frameworks is called *lenses* which focus on the view-update problem [13]. In this paper, whenever bidirectional transformation is mentioned, we refer to the *lenses* framework.

A BX consists of a pair of transformations: a forward transformation named *get*, which extracts a part of information from a *source* to construct the *view*, and a backward transformation named *put*, which takes the *source* and an updated *view* as input to produce an updated *source* embedding information from the updated *view* [14]. The pair of transformations should be *well-behaved*, i.e., they should satisfy two *round-tripping* laws which are defined as follows:

$$put\ s\ (get\ s) = s \qquad (\textit{GetPut}\text{ law})$$
$$get\ (put\ s\ v) = v \qquad (\textit{PutGet}\text{ law})$$

where s and v respectively indicate a *source* and a *view*.

In this work, we adopt a putback-based bidirectional programming language called BIGUL (for Bidirectional Generic Update Language), which is formally verified in the dependently typed programming language AGDA [15], [16] to guarantee that any putback transformation written in BiGUL is well-behaved [9]. And for use in practical applications, BIGUL is ported to Haskell. Following the putback-based approach, a BIGUL program can be evaluated as either a *put* or a *get* by only describing the *put* function of bidirectional transformations.

At present, BIGUL provides seven basic programs called *constructors*: `Skip`, `Replace`, `Prod`, `RearrV`, `RearrS`, `Case` and `Fail`. The semantic of `Skip`'s *get* function is take anything as input (*source*) and output nothing (*view*). Its *put* function is remaining the *source* unchanged. The semantic of `Replace`'s *get* function is output what is inputted. And the *put* function ignore the *source* and return the *view*. `Prod` uses two sub programs, `bx1` and `bx2`, to transform between two pairs. `bx1` is reponsible for the first elements of the pairs, and `bx` is responsible for the second ones.

Here is a simple BIGUL program:

```
rplFst :: BiGUL (Int,Int) (Int, ())
rplFst = Prod Replace Skip
```

The *source* is a pair of `Int` variables, and the *view* is an `Int` variable with an empty unit. The first elements of both pairs are associated by `Replace`, while the second by `Skip`. There are two running examples for this BIGUL program which respectively show the behavior of both *get* and *put* function:

```
>> get rplFst (3,5)
Right (3,())

>> put rplFst (3,5) (9,())
Right (9,5)
```

The `Right` in the result means no error occurs, otherwise it will be a `Left`.

As the sample program shows, *constructors* can be composed to constitute more complicated BIGUL programs. To see the introduction of the rest *constructors* and the other details, as well as the latest updates, about BIGUL, please refer to [17].

IV. Framework for synchronization

We propose a framework, showing in Fig. 6, to synchronize between feature configurations and use cases, i.e., a feature configuration can be automatically updated with an adjusted use case, and vise versa. This work is based on our previous work [5], called TDL, which focuses on specifying the transformation information used for automated derivation of use cases.

The framework consists of three BXs implemented in BIGUL: 1) *sync_FC_guards* is responsible for synchronizing a feature configuration with a set of guards that belong to active X-rules; 2) *sync_guards_activities* takes responsibility for the synchronization between guards and the corresponding activities according to the rules; 3) *sync_activities_UC* synchronizes a use case with a set of activities by rearranging them into a proper order. The three sychronizations are all based on the knowledge provided by the feature model and the X-rules.

A. Knowledge base

We treat a feature configuration as a set of features that conforms to the constraints imposed on them in the feature

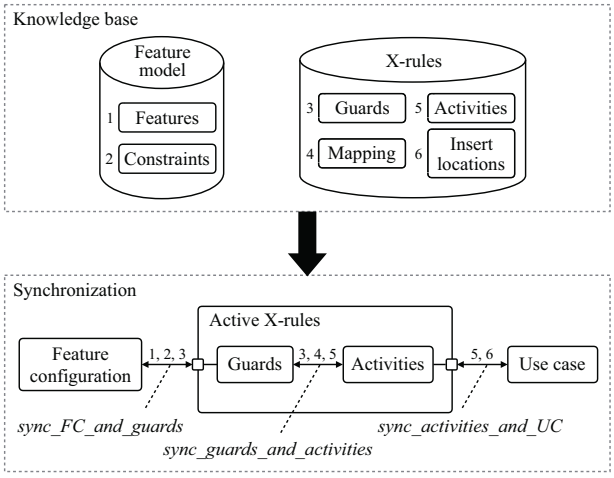

Fig. 6: Framework of the approach

model. Here we use the constraints to stand for both the explicit constriants and constraints implicated by the refinements in the feature model.

X-rules, which related with a paticular use case are decomposed into *guards*, *mapping*, *activities* and *insert locations*.

- *Guards* consists of all the related X-rules's guards. Since each guard is a conjunction of $features$ and $\neg features$, as the syntax showed in Fig. 5, the $\wedge$ operators can be omitted, and the guard can be denoted as a set of $features$ and $\neg features$.
- *Mapping* denotes the relations between *guards* and *activities*. Each record of *Mapping*, which takes the form of `(Guard, Activity)`, indicates when the *guard* is true, the *activity* has to be inserted into the use case.
- *Activities* includes all the activities that can be inserted into the particular use case. The activities are declared in Haskell as follows:

```
type Actor = String
type Behavior = String
type Act = (Actor, Behavior)
data Activity = Single Act
              | Branch Act [Activity] [Activity]
              | Loop Act [Activity]
```

A `Single` activity is a tuple `(Actor, Behavior)`, with `Actor` denotes the excutor of the `Behavior`, like *Click check out* (A1) (TABLE Ia, Fig. 3a) is a basic behavior, and *User* is its `Actor`.

A `Branch` activity is a compound activity with two branches. The activity is a decision node followed with two conditions. Each condition leads a branch that is a sub-list of activities. In Fig. 3b, *Pick a payment method* (B1) is the dicision activity. One of the conditions is *Credit card* (C2) followed with *Input credit card info* (A4), and the other one is *Cash on delivery* (C3) followed with an empty branch.

A `Loop` activity denotes an iteration. It consists of a decision activity, a condition, and a loop body that is also a sub-list of activities. *Check if the address is in scope* (L1), *False* (C1) and *Input address info* (A3) constitute a simple `Loop` activity in Fig. 3b.

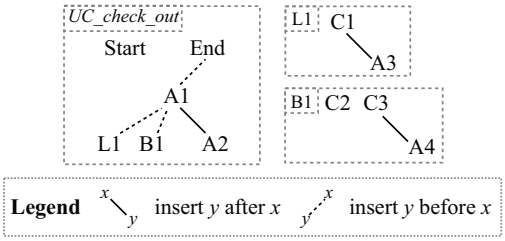

Fig. 7: Insert locations of *UC_check_out*

- *Insert locations* records the relative position information of insert operations in X-rules (TABLE Ib). For each activity, there are only one operation that is responsible for its insertion, and one unique corresponding record in *insert locations*.

We further form *insert locations* into tree structure. If the insert operation of feature x is "Insert x after/before x'", then x' is the parent of x in the *insert locations* trees. The relative positions are denoted with different type of lines: solid lines for "after" and dashed lines for "before". We define *tree* x denotes the sub-tree of *insert locations* trees where x is the root node of the sub-tree. The compound activity is denoted as one node in the trees. The *insert locations* trees of its subordinate activities are built seperately. One thing to note is that the subordinate activities are regarded as descendants of the compound activity as well in this work. Fig. 7 shows the *insert locations* trees of *UC_check_out*. Activity A1 should inserted before *End*. L1 and B1 should inserted before A1, while A2 should inserted after A1. The subordinate activities of L1 and B1, including C1, A3, C2, etc, form the sub-trees respectively in the right part of Fig. 7.

The *insert locations* trees satisfy the following properties:

- ***Completeness***: All the activities in the trees, except *Start* and *End*, are in the set of *activities*, and vice versa;
- ***2Roots***: Each activity in *activities* is a descendant of either *Start* or *End*;
- ***Reverse Heredity***: If an activity is in a use case, then its parent has to be in the use case as well; (This property can be generalized to all its ancestors.)
- ***Heredity***: Assuming activity x should be inserted after(/before) its parent x', then all the descendants of x will be after(/before) x' in the use case if they are included;
- ***continuity***: Given a valid use case, activities belong to the same sub-tree are continuous in the use case. In other words, for any two activities x and y in the use case, assuming they both belong to *tree* z, then all the activities between x and y in the use case all belong to *tree* z.

B. Synchronization between feature configuration and guards

Given a valid feature configuration, the corresponding set of guards in the active X-rules (abbr. to *active_guards*) is determined. At the same time, the *active_guards* can be extracted from the feature configuration. Accordingly, the feature model and the *active_guards* can be regarded as the source and view in the bidirectional model transformation, and the synchronization between them can be implemented in BIGUL.

BIGUL is in nature a putback-based language, which enables us to describe only the behavior of the backward transformation, i.e., updating the original feature configuration with updated *active_guards*.

State is a boolean attribute of a feature, which denotes whether the feature is selected (*true*) or removed (*false*) in a configuration. A feature configuration is an assignment to the *state* of all features in a feature model. Assuming *fm* is the set that contains all the features in the feature model, we define the following sets:

$$Select = \{x | x \in fm,\ \ x.state = true\}$$
$$Remove = \{x | x \in fm,\ \ x.state = false\}$$

Both sets are initialized to empty sets. Noticing that all *active_guards* have to be true, in the sense that each feature appearing in these guards needs to be selected in the updated feature configuration. These features are inserted into *Select*.

Select and *Remove* are iteratively constructed depending on the *require* relations, the *exclude* relations and the inactive guards (at least one feature in an inactive guard should be removed) until no more features can be added in *Select* or *Remove* until the *state* of features that belong to either sets are determined.

The *state* of rest features are enumerated. In order to update the feature configuration with minimal changes, when we traverse a feature, we try its origin *state* first. For instance, feature *real goods* is in the original configuration showed in Fig. 2. When we traverse it, we first try to add it into *Select* first. As a special case, if the updated *active_guards* haven't changed at all, it which guarantee that the feature configuration will stay the same through the transformation.

It's worth mentioning that a use case only related to a small part of the whole feature model and features irrelevant to the updated use case can usually keep their original binding states. Hence, the enumeration usaually stop quickly and can keep a low cost in average, though it's not efficient in theory.

The synchronization is defined in Haskell as:

```
sync_FC_guards :: BiGUL [Feature] [[Feature]]
```

Here's an example. The configuration showed in Fig. 2 will be represented by a sorted `List` of features:

```
fc = [address, check_out, credit_card,
      delivery_range, goods, online_store,
      payment, real_goods, shipping]
```

We use a sorted `List` of `Features` to indicate the feature configuration intead of a `Set` of `Features` because BIGUL supports `List` better.

Applying the *get* function of `sync_FC_guards` to `fc`, we can get the corresponding *active_guards*:

```
ag = get sync_FC_guards fc
   = [[check_out], [credit_card], [payment],
      [shipping]]
```

Each list in the result represent an *active_guard*.

If a new guard `[cash_on_delivery]` is added into the *active_guards*, the feature configuration can be updated with the *put* function of `sync_FC_guards`:

```
ag' = [[cash_on_delivery], [check_out],
       [credit_card], [payment], [shipping]]
fc' = put sync_FC_guards fc ag'
    = [address, cash_on_delivery, check_out,
       credit_card, delivery_range, goods,
       online_store, payment, real_goods,
       shipping]
```

C. Synchronization between guards and activities

Each X-rule consists of a unique guard and a sequence of operations. And the insert operation of each activity only appear once in the whole set of X-rules. Therefore, *mapping*, which denotes the relation between guards and activities, is an 1-to-n mapping.

Whereas, the mapping between the set of guards whose values are true (*active_guards*) and the set of activities which should be inserted into the use case (*active activities*) is an 1-to-1 mapping. This is because when a guard is true, it will activate the X-rule and all the corresponding activities should be inserted into the use case.

Since the mapping is 1-to-1, either side can be regarded as the source of the BIGUL program. We take *active activities* as source and *active_guards* as view in this framework. The behavior of backward transformation simply follows the mapping relationship from guards to activities according to *mapping*.

The synchronization is defined in Haskell as:

```
sync_guards_activities ::
    BiGUL [Activity] [[Feature]]
```

Here's an example.

```
act = [A1, A2, A3, A4, B1, C1, C3, End, L1, Start]
```

where `act` represents the activities, including branches, loops and conditions, which should be inserted into the use case *UC_check_out*. We also use a sorted `List` to indicate a `Set` here.

Applying the *get* function of `sync_guards_activities` to get the corresponding *active_gurads*:

```
ag = get sync_guards_activities act
   = [[check_out], [credit_card], [payment],
      [shipping]]
```

Just like the example in Section IV-B, once we add `[cash_on_delivery]` into `ag`, the activities could be updated by the *put* function of `sync_guards_activities`:

```
ag' = [[cash_on_delivery], [check_out],
       [credit_card], [payment], [shipping]]
act' = put sync_guards_activities act ag'
     = [A1, A2, A3, A4, B1, C1, C2, C3,
        End, L1, Start]
```

D. Synchronization between activities and use case

Given a set of activities and a use case, we split the activity set according to whether each activity belongs to *tree Start* or *tree End* in the *insert locations* trees. The use case, which is a sequence of activities, is splitted in the similar way. According to the *continuity* property, the two sub-sequences are both continuous in the original use case.

We synchronize the subsets and the corresponding sub-sequences respectively with a BIGUL program called *sub_sync*, which is responsible for updating a sub-sequence with the corresponding subset. They both correspond to the same subtree of *insert locations* (assuming *x* is the root of the subtree). The behavior of *sub_sync* is defined as follows:

- *If the subset is empty*: We remove the whole sub-sequence from the use case;
- *If the sub-sequence is empty*: We insert *x* into the sub-sequence and call *sub_sync* again.
- *If the subset and the sub-sequence are not empty*: In this case, *x* has to be in both of them (according to *P3*). We first extract *x* respectively from the subset and the sub-sequence. If *x*'s data structure is:
  - `Single`: There is nothing need to be done;
  - `Branch`: We apply *sub_sync* to each branch with the corresponding sub-subset extracted from the subset;
  - `Loop`: Just like `Branch`, we apply *sub_sync* to the loop body with the corresponding sub-subset extracted from the subset.

  Then we split the subset, except *x*, into sub-subsets according to child-subtrees of *x*, where a child-subtree of *x* means a subtree whose root is a child of *x*. For instance, if *x* has two children *y* and *z*, then the subset will be splitted in to two sub-subsets. One sub-subset contains the activities that belong to *tree y*. The rest activities, which have to be descendants of *z*, are in the other sub-subset. The sub-sequence is splitted into sub-sub-sequences in the same way. We recursively apply *sub_sync* to the pairs of corresponding sub-subsets and sub-sub-sequences.

The synchronization is defined in Haskell as:

```
sync_activities_UC :: BiGUL [Activity] [Activity]
```

Given a use case, for instance, the one showed in Fig. 3a can be represented as:

```
uc = [Start, (L1, [C1, A3]), (B1, [], [C3, A4]),
      A1, A2, End]
```

Since `B1` has only one branch in this case, Fig. 3a omits `B1` as well as `C3`.

With the *get* function of `sync_activities_UC`, we can get the corresponding sorted list of activities:

```
act = get sync_activities_UC uc
    = [A1, A2, A3, A4, B1, C1, C3, C4, L1, Start]
```

If `C2` is added into `act`, we should find a proper position for it in `uc`. The *put* function of `sync_activities_UC` is responsible for that:

```
act' = [A1, A2, A3, A4, B1, C1, C2, C3, End, L1,
        Start]
uc' = put sync_activities_UC uc act'
    = [Start, (L1, [C1, A3]), (B1, [C2], [C3, A4]),
       A1, A2, End]
```

E. Composing the BiGUL programs

BIGUL provides a *constructor* called `Compose`:

```
Compose :: BiGUL s u -> BiGUL u v
   -> BiGUL s v
```

which is used to compose two BIGUL program together. It requires that the first BIGUL program's view type is same as the second one's source type. By using `Compose`, *sync_guards_activities* and *sync_activities_UC* can be combined to:

```
sync_guards_UC :: BiGUL [Activity] [[Feature]]
sync_guards_UC = Compose
    sync_activities_UC sync_guards_activities
```

Since the *view* type of *sync_FC_guards* is different from the *source* type of *sync_guards_UC*, `Compose` can not be used to compose *sync_FC_guards* and *sync_guards_UC*.

However, *sync_FC_guards* and *sync_guards_UC* have the same *view* type. In this paper, we construct a new composing operation, `ComposeSVS`, to compose two BXs that share the same view. Assuming the two BXs are:

```
bxL :: BiGUL SourceL View
bxR :: BiGUL SourceR View
```

where `SourceL` is the *source* type of `bxL`, and `SourceR` is `bxR`'s. The two BIGUL program share the same *view* type, `View`.

The forward and backward transfomations of them are respectively named: *getL*, *putL* and *getR*, *putR*:

```
putL = put bxL
getL = get bxL
putR = put bxR
getR = get bxR
```

We define the result of `ComposeSVS bxL bxR` has a type of `BiGULSVS`, and it has these two transformations:

$$putR2L(sl, sr) = putL(sl, getR(sr)) \quad \text{(def1)}$$

$$putL2R(sr, sl) = putR(sr, getL(sl)) \quad \text{(def2)}$$

where $sl \in SourceL$, $sr \in SourceR$.

The two transformations are no longer satisfied with the defination of *get* and *put* in *lenses* framework. We define the new consistency and the round-tripping laws of `BiGULSVS` as follows:

$$sl \sim sr \triangleq getL(sl) = getR(sr) \quad \text{(cons)}$$

$$sl \sim sr \Rightarrow putR2L(sl, sr) = sl \quad \text{(law1.1)}$$

$$sl \sim sr \Rightarrow putL2R(sr, sl) = sr \quad \text{(law1.2)}$$

$$putR2L(sl, sr) \sim sr \quad \text{(law2.1)}$$
$$putL2R(sr, sl) \sim sl \quad \text{(law2.2)}$$

Proof of (law1.1):

$$\begin{aligned} & putR2L(sl, sr) \\ &= putL(sl, getR(sr)) && \text{(def1)} \\ &= putL(sl, getL(sl)) && \text{(cons)} \\ &= sl && (\textit{GetPut}\text{ law}) \end{aligned}$$

Proof of (law2.1):

$$\begin{aligned} & getL(putR2L(sl, sr)) \\ &= getL(putL(sl, getR(sr))) && \text{(def1)} \\ &= getR(sr) && (\textit{PutGet}\text{ law}) \\ &\Leftrightarrow putR2L(sl, sr) \sim sr && \text{(cons)} \end{aligned}$$

law1.2 and law2.2 are respectivly isomorphic with (law1.1) and (law2.1).

The proof of $putR2L$ and $putL2R$'s well-behavedness guarantee that our framework, a composition of *sync_FC_guards*, and *sync_guards_UC*, is likewise well-behaved.

Accordingly, we can define:

```
sync_FC_UC :: BiGULSVS [Feature] [Activity]
sync_FC_UC = ComposeSVS
                sync_FC_guards sync_guards_UC
```

`sync_FC_UC` describes the BX between a feature configuration and a use case. For instance, we can use `putL2R` to transform the feature configuration showed in Fig. 2 to the use case showed in Fig. 3a:

```
fc = [address, check_out, credit_card,
      delivery_range, goods, online_store,
      payment, real_goods, shipping]
uc = putL2R sync_FC_UC fc []
   = [Start, (L1, [C1, A3]), (B1, [], [C3, A4]),
      A1, A2, End]
```

And if `uc` is changed, like Fig. 3b, into:

```
uc' = [Start, (L1, [C1, A3]), (B1, [C2],
       [C3, A4]), A1, A2, End]
```

we can use `putR2L` to update the feature configuration:

```
fc' = putR2L sync_activities_UC fc uc'
    = [address, cash_on_delivery, check_out,
       credit_card, delivery_range, goods,
       online_store, payment, real_goods,
       shipping]
```

V. Threats to validity

A. Internal Validity

A feature configuration usually corresponds more than one use case. After synchronizing the configuration with one particular updated use case, the updated configuration may be no longer satisfied with the other use cases. Since this paper focuses on the synchronization between a feature configuration with a use case, we argue that only adjusting one use case at a time should be allowed. After updating the feature configuration, all the rest use cases should be synchronized then. The impacts on other use cases should be confirmed by stakeholders.

B. External Validity

The synchronization requires the validity of feature configurations and use cases. Inputting an invalid use case will cause error. The validity of a feature configuration can be checked through model checking techniques and tools. Whereas, there's no explicit constraints between activities.

VI. Related Works

Griss et al. [2] and Bragança et al. [18] propose high-level guidelines to derive feature models from use cases. Mefteh et al. [19] [20] propose an approach to extract feature models from documented use cases. Their approaches focus on the derivation from use cases to feature models, and leave no formal specified transformation information to support the bidirectional transformation. In their work, a feature corresponds to a use case in general, but the variability inside a use case does not reflect in the feature model. Besides, Griss et al. and Bragança et al.'s guildlines are based on extended use cases which can represent commonalities and variabilities, and Mefteh et al.'s work is based on documented use cases with detailed description of *goal in context*. Our approach, which is based on TDL, does not require use cases with extensions or extra description.

Cazrnecki and Antikiewicz [4] propose a general template-based approach for mapping feature configurations to other models, including UML activity diagrams. A drawback of this approach is that the transformation information is not rigorously specified in a formal way. And transformation from other models to feature configurations is beyond the method's capability.

Bonifcio et al. [3] and our previous work [5] propose approaches to construct transformation information between feature models and use cases. The information enables the derivation from feature configurations to use cases. The main limitation of these works is that they only provide unidirectional transformations. To be more specific, transformation information written in TDL implies the variabilities about the order of activities in use cases. That means, given a feature configuration, variant use cases with the same activities but in different orders can be derived by applying the insert operation in different orders. However, the derivation method proposed in our previous work apply insert operations in a default order, ignoring other posibilities. We improve our previous work by supporting the bidirectional transformation between feature configurations and use cases, and allowing derivating use cases with all valid insert oder.

Eriksson et al. [21] propose an approach that marrying features and use cases. In their approach, features, use cases and change cases are integrated into a coherent two-layer product line requirements model. They also propose an approach to construct transformation information between feature model and use case [The PLUSS Approach - Domain Modeling with Features, Use cases, and Use case realizations]. However, in both approaches, feature model is just a tool for visualizing commonalities and variabilities of use case. That is to say,

feature models in these approaches are feature models only in syntax, but not in semantics.

Wang et al. [22] propose a use case based approach for improving the evolution of an existing portfolio of products into a software product line by mining the requirements specifications of existing valid product configurations and automatically creating a feature model. A main drawback of the work is no formal transformation information is produced with the process of feature models construction.

Hajri et al. [6] [7] propose a product line methodology centred around use case modeling, called PUM, for documenting variability in use case diagrams and specifications. A main drawback of this work is the grain of the variability, which is not fine enough. It is capable of describing whether a use case should be in or out of the use case diagram. But it can't describe the variability inside a use case. However, one use case may behave differently in variant products.

Snchez et al. [23] and Zschaler et al. [24] propose a generative approach to building a family of languages for specifying the relationship between variability models and other models in software product line engineering. The approach can be applied to derive use case models of a specific product from a feature configuration. One of the drawback is similar with Hajri et al.'s work, which is it only support the variability of use case models on the use case level. It can't describe the variability inside the use cases. The other drawback is that it can't update the feature configuration with an updated use case diagram.

VII. Conclusion and future work

In this paper, we propose a TDL based framework for synchronization between feature configurations and use cases through bidirectional programming (BIGUL). Our framework provides a well-behaved bidirectional transformation composed of three BIGUL programs. It allows changes, either in feature configurations or use cases, propagate to the other side synchronically.

Our future work will focus on following three points:

1) Developing a mechanism for cheking the validity of use cases and ensuring stakeholders to adjust the use case in the correct range;
2) Evolving mechanisms for feature model when stakeholders ask for adding some activities out of the scope of the original domain model;
3) Supporting synchronization feature configurations with other kinds of reusable software asset, for example, the class diagram model, the software architecture model, and also the source code.

Acknowledgment

The authors would like to thank Dr. Hsiang-Shang Ko from the National Institute of Informatics, Tokyo, Japan, as well as Mr. Tao Zan, Mr. Zirun Zhu and Mr. Yongzhe Zhang, from Sokendai, Tokyo, Japan, for their help with BIGUL development.

This work is supported by Science Fund for Creative Research Groups of the National Natural Science Foundation of China (Grant No. 61121063), National Natural Science Foundation of China (Grant Nos. 61272163, 91318301), and Internship Program of the National Institue of Informatics of Japan.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.

[2] M. L. Griss, J. Favaro, and M. D. Alessandro, "Integrating feature modeling with the rseb," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE, 1998, pp. 76–85.

[3] R. Bonifcio and P. Borba, "Modeling scenario variability as crosscutting mechanisms," in *International Conference on Aspect-Oriented Software Development, Aosd 2009, Charlottesville, Virginia, Usa, March*, 2009, pp. 125–136.

[4] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *International conference on generative programming and component engineering*. Springer, 2005, pp. 422–437.

[5] W. Yu, W. Zhang, H. Zhao, and Z. Jin, "Tdl: a transformation description language from feature model to use case for automated use case derivation," in *International Software Product Line Conference*, 2014, pp. 187–196.

[6] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach," in *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2015.

[7] ——, "Configuring use case models in product families," *Software & Systems Modeling*, pp. 1–33, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10270-016-0539-8

[8] J. N. Foster, "Bidirectional programming languages," 2010.

[9] H.-S. Ko, T. Zan, and Z. Hu, "Bigul: a formally verified core language for putback-based bidirectional programming," in *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 2016, pp. 61–72.

[10] W. Zhang, H. Mei, and H. Zhao, "Feature-driven requirement dependency analysis and high-level software design," *Requirements Engineering*, vol. 11, no. 3, pp. 205–220, 2006.

[11] "OMG unified modeling language ™ (OMG UML) Version 2.5," 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/PDF

[12] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *Theory and Practice of Model Transformations*. Springer, 2009, pp. 260–283.

[13] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, p. 17, 2007.

[14] Z. Hu, A. Schurr, P. Stevens, and J. F. Terwilliger, "Dagstuhl seminar on bidirectional transformations (BX)," *ACM SIGMOD Record*, vol. 40, no. 1, pp. 35–39, 2011.

[15] U. Norell, *Towards a practical programming language based on dependent type theory*. Citeseer, 2007, vol. 32.

[16] ——, *Dependently Typed Programming in Agda*. Springer Berlin Heidelberg, 2009.

[17] "Principle and practice of bidirectional programming in BIGUL," 2016. [Online]. Available: http://www.prg.nii.ac.jp/project/bigul/tutorial.pdf

[18] A. Bragança and R. J. Machado, "Automating mappings between use case diagrams and feature models for software product lines," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, 2007, pp. 3–12.

[19] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Feature model extraction from documented uml use case diagrams," *ADA USER*, vol. 35, no. 2, p. 107, 2014.

[20] ——, “Implementation and evaluation of an approach for extracting feature models from documented uml use case diagrams,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1602–1609.
[21] M. Eriksson, J. Brstler, and K. Borg, “Marrying features and use case for product line requirements modeling of embedded systems,” pp. 73–83, 2004.
[22] B. Wang, W. Zhang, H. Zhao, Z. Jin, and H. Mei, “A use case based approach to feature models’ construction,” in *IEEE International Requirements Engineering Conference, RE*, 2009, pp. 121–130.
[23] P. Snchez, N. Loughran, L. Fuentes, and A. Garcia, “Engineering languages for specifying product-derivation processes in software product lines,” in *Software Language Engineering*, 2008, pp. 188–207.
[24] S. Zschaler, P. Snchez, J. Santos, M. Alfrez, A. Rashid, L. Fuentes, A. Moreira, J. Arajo, and U. Kulesza, “Vml* a family of languages for variability management in software product lines,” in *International Conference on Software Language Engineering*, 2010, pp. 82–102.