Identifying and Visualising Commonality and Variability in Model Variants*

Jabier Martinez[1][2], Tewfik Ziadi[2], Jacques Klein[1], and Yves le Traon[1]

[1] SnT, University of Luxembourg
Luxembourg, Luxembourg
`(jabier.martinez|jacques.klein|yves.letraon)@uni.lu`
[2] LIP6, Université Pierre et Marie Curie
Paris, France
`tewfik.ziadi@lip6.fr`

Abstract. Models, as any other software artifact, evolve over time during the development life-cycle. Different *versions* of the same model are thus existing at different times. Model comparison of different versions has received a lot of attention in recent years. However, existing techniques focus on comparing only two model versions at the same time to identify model differences. Independently of model versioning context, another dimension of variation, called *variation in space*, appears in models. Contrary to *variation in time*, variation in space means that a set of model *variants* exists and should be maintained. Comparing all these model variants to identify common and variable elements becomes thus a major challenge. Current approaches for model variants comparison lack of flexibility and appropriate visualisation paradigm. The contribution of this paper is the Model Variants Comparison approach (MoVaC). This approach compares a set of model variants and identifies both commonality and variability in the form of what is referred to as *features*. Each feature consists in a set of atomic model-elements. MoVaC also visualizes the identified features using a graphical representation where common and variable features are explicitly presented to users. We validate the approach on two use cases demonstrating the flexibility of MoVaC to be applied to any kind of EMF-based model variants.

1 Introduction

One of major challenges in model-driven engineering is model comparison and it has received a lot of attention in recent years. Indeed, models, as any other software artifact, evolve over time during the development life-cycle. Different *versions* of the same model are thus existing at different times. Moreover, since different models may be capturing different viewpoints of a system, model comparison is becoming particularly relevant in industrial contexts where different developers work on the same model [7]. In addition, model comparison is an

* Preprint

enabler for complex operations such as merging, model compositions and model transformations.

The first generation of comparison techniques mainly focused on a single kind of models such as UML models [1,10,13]. However, in recent years, and motivated by the success of the Eclipse Modeling Framework (EMF) [4], generic approaches have been proposed to compare any kind of EMF-based models. EMF Compare [3] and EMF DiffMerge [5] are examples of such platforms.

All these generic approaches are limited to a comparison between two versions only. This limitation can be explained because, historically these approaches have been inspired by classical software version control systems [8] where the main issue is to manage *variation in time*. Two program versions are thus analysed to identify differences and eventually merge them. However, with the emergence of software product lines engineering [16], another dimension of variation, called *variation in space*, appears in models. Contrary to variation in time, variation in space means that a set of model *variants* exists and that all these variants should be maintained. Model variants cohabit and evolve at the same time, each of them addressing some new specific requirement of the system, some dedicated features or functionality. Thus the models life-cycle is no longer linear, but it is a tree with parallel variant branches. The problem is that no instrument exists to capture this tree-like evolution of model variants. Comparing all these model variants to identify common and variable elements become thus a major challenge as the interest is to be able to compare more than two models. The objective of this comparison is to eventually refactor these model variants to adopt a software product line approach.

Another observation in existing model comparison platforms is that they only focus on the identification of differences between two model versions but the common part is not explicitly identified in the result of the models comparison. For instance, when we use EMF Compare to compare two model versions, the displayed result only highlights the differences. Common elements can be obtained implicitly by manipulating the result calculated by EMF Compare [3] or the comparison result of EMF DiffMerge. However, this information is not presented explicitly. This also can be justified by the fact that control version systems only aim identifying differences to apply merge. In variation in space it is of high relevance to present explicitly the commonality of the set of model variants as this represents the core that is shared by all model variants.

This paper proposes a new approach for model comparison that can be applied to a set of models variants instead of binary comparison of two models. The approach is called **Mo**del **Va**riants **C**omparison (MoVaC). Beyond offering a systematic and semantically well founded comparison method, another interest is to contribute a generic approach that can be applied for any EMF based models and also to provide a visualisation paradigm of the obtained result.

The rest of paper is organized as follows: Section 2 discusses related work while Section 3 presents an illustrative example. Section 4 presents our approach and applies it on the example. Section 5 presents the validation of the approach and Section 6 concludes this work and presents some perspectives.

2 Related Work

Stephan et al. [19] and *Kolovos et al.* [14] present complete surveys on model comparison and a classification of existing approaches. They showed that several definitions for model comparison exist. In this paper, we consider model comparison as presented by *Brun et.al.* [3]. In this context, model comparison is decomposed in two main steps:

- *Calculation*: In this first step, a procedure, a method or an algorithm is proposed to compare models.
- *Representation and Visualisation*: The outcome of the calculation can be represented in some form and a visualisation is displayed to users.

Existing comparison approaches differ by the way they consider these two steps. The approaches for managing model versions, such as EMF Compare [3] and EMF DiffMerge [5], implement the two steps. EMF Compare uses various statistics and metrics to calculate the match score in the calculation step and the result of the comparison is presented by means of models. EMF DiffMerge is based on customizable Match and Diff policies. However, and as mentioned above, the main observation concerning these approaches is that they only compare two model versions at the same time. If we consider the variation in space dimension where a set of model variants coexist at the same time, these approaches do not allow comparing all these model variants together. EMF Compare and EMF DiffMerge also lack in explicitly highlighting commonality.

Independently from model versioning, the problem of comparing model variants is well known in Software Product Line Re-engineering [9] where the main issue is to analyse a set of model variants to identify commonality and variability. Some of the approaches in this area are related to a specific kind of artifacts and therefore are not generic. *Ryssel et al* [18] compare model variants that are represented as function-blocks. *Ziadi et al.* [21] propose an approach to analyse the source code through the use of UML class diagrams of a set of software variants and identify commonality and variability between them. These approaches are not generic and they do not provide a visualisation paradigm.

Rubin et al. [17], study the problem of model variants comparison mainly in the context of UML models (UML statecharts). They compare a set of UML statecharts with the goal to refactor them as a product line. The authors use XMI principles to represent model variants. They thus justify that their approach can be applied for any kind of meta-models. However, no visualisation paradigm is provided. Indeed, they are focusing on merging input product variants into a generic model and not on highlighting commonality and variability between model variants.

Existing approaches in the context of Software Product Line Re-engineering compare a set of model variants at the same time. However, they only consider the Calculation step. The Representation and Visualisation step is not considered. This is because their main objective concerns refactoring model variants into a product line model without including a domain expert in the process.

We claim that the domain expert must take part in the analysis of the mined commonality and variability information for product line adoption.

In this paper we propose the MoVaC approach that allows comparing a set of model variants at the same time to identify commonality and variability between them. Beyond offering a meta-model independent calculation step, our approach provides a graphical visualisation of the comparison result using the concepts of Features. The approach will be presented in Sect. 4.

3 Illustrative Example

As a concrete example, we use throughout this paper a set of UML model variants representing a set of banking systems [21]. Each model variant represents a simple banking application. The variation between these model variants is related to: limit on the account, consortium entity, and to the currency exchange, which are only present in some variants. Figure 1 illustrates the eight model variants that we consider for comparison. The differences between these model variants concern the presence, or absence, of some classes, attributes and/or operations. The first `Product1Bank` UML model represents a full model. It contains all model elements to support limit on account, consortium entity, and currency exchange. On the contrary, in the `Product2Bank` model, the `Account` class is defined without the limit and currency attributes. We also note the absence of `Converter` and `Consortium` classes. The `Product3Bank` model is defined with information related to currency exchange and consortium but without all elements related to the limit capacity.

In next section, we apply our comparison approach on these model variants to identify commonality and variabilities.

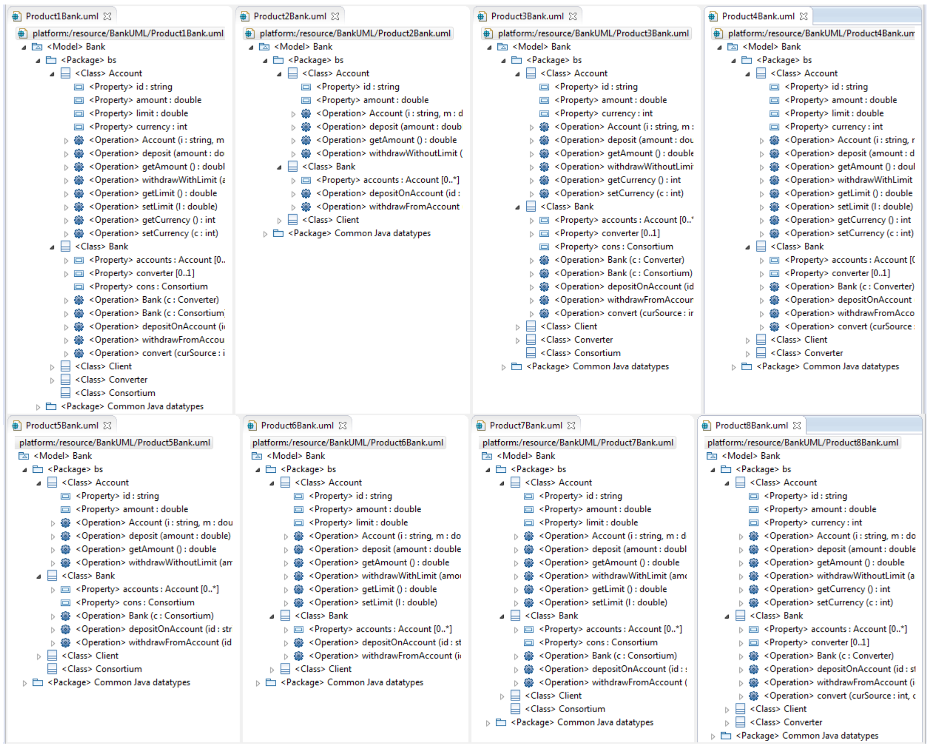
4 Model Variants Comparison Approach

MoVaC [1] is a meta-model independent approach which compares a set of model variants to identify the commonality and the variability between them. To achieve this comparison, our approach follows the general framework of model comparison presented in Sect. 2. It thus consists in two main steps: The *Calculation* and *Representation and Visualisation* steps. Next subsections present each step.

4.1 Step 1: Calculation

As mentioned above, the calculation step in model comparison approaches consists in defining an algorithm that identifies commonality and variability between the model variants. The main idea of MoVaC is to divide each model variant into a Set of Atomic Model-Elements (SoAMEs). Model comparison between a set of model variants is then defined as an equivalence relation between their SoAMEs. This step starts by dividing each model variant into a SoAMEs. Each Atomic

[1] The source-code of MoVaC could be found at https://bitbucket.org/jabi/but4reuse



```
Product1Bank.uml
platform:/resource/BankUML/Product1Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> limit : double
        <Property> currency : int
        <Operation> Account (i : string, m
        <Operation> deposit (amount : do
        <Operation> getAmount () : doubl
        <Operation> withdrawWithLimit (a
        <Operation> getLimit () : double
        <Operation> setLimit (l : double)
        <Operation> getCurrency () : int
        <Operation> setCurrency (c : int)
      <Class> Bank
        <Property> accounts : Account [0..
        <Property> converter [0..1]
        <Property> cons : Consortium
        <Operation> Bank (c : Converter)
        <Operation> Bank (c : Consortium
        <Operation> depositOnAccount (i
        <Operation> withdrawFromAccou
        <Operation> convert (curSource : i
      <Class> Client
      <Class> Converter
      <Class> Consortium
    <Package> Common Java datatypes
```

```
Product2Bank.uml
platform:/resource/BankUML/Product2Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Operation> Account (i : string, m : d
        <Operation> deposit (amount : doub
        <Operation> getAmount () : double
        <Operation> withdrawWithoutLimit (
      <Class> Bank
        <Property> accounts : Account [0..*]
        <Operation> depositOnAccount (id :
        <Operation> withdrawFromAccount
      <Class> Client
    <Package> Common Java datatypes
```

```
Product3Bank.uml
platform:/resource/BankUML/Product3Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> currency : int
        <Operation> Account (i : string, m :
        <Operation> deposit (amount : dou
        <Operation> getAmount () : double
        <Operation> withdrawWithoutLimi
        <Operation> getCurrency () : int
        <Operation> setCurrency (c : int)
      <Class> Bank
        <Property> accounts : Account [0..*
        <Property> converter [0..1]
        <Property> cons : Consortium
        <Operation> Bank (c : Converter)
        <Operation> Bank (c : Consortium)
        <Operation> depositOnAccount (id
        <Operation> withdrawFromAccour
        <Operation> convert (curSource : ir
      <Class> Client
      <Class> Converter
      <Class> Consortium
    <Package> Common Java datatypes
```

```
Product4Bank.uml
platform:/resource/BankUML/Product4Bank.um
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> limit : double
        <Property> currency : int
        <Operation> Account (i : string, r
        <Operation> deposit (amount : d
        <Operation> getAmount () : doul
        <Operation> withdrawWithLimit
        <Operation> getLimit () : double
        <Operation> setLimit (l : double)
        <Operation> getCurrency () : int
        <Operation> setCurrency (c : int)
      <Class> Bank
        <Property> accounts : Account [(
        <Property> converter [0..1]
        <Operation> Bank (c : Converter)
        <Operation> depositOnAccount
        <Operation> withdrawFromAcco
        <Operation> convert (curSource
      <Class> Client
      <Class> Converter
    <Package> Common Java datatypes
```

```
Product5Bank.uml
platform:/resource/BankUML/Product5Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Operation> Account (i : string, m : dou
        <Operation> deposit (amount : double)
        <Operation> getAmount () : double
        <Operation> withdrawWithoutLimit (am
      <Class> Bank
        <Property> accounts : Account [0..*]
        <Property> cons : Consortium
        <Operation> Bank (c : Consortium)
        <Operation> depositOnAccount (id : str
        <Operation> withdrawFromAccount (id
      <Class> Client
      <Class> Consortium
    <Package> Common Java datatypes
```

```
Product6Bank.uml
platform:/resource/BankUML/Product6Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> limit : double
        <Operation> Account (i : string, m : do
        <Operation> deposit (amount : double
        <Operation> getAmount () : double
        <Operation> withdrawWithLimit (amo
        <Operation> getLimit () : double
        <Operation> setLimit (l : double)
      <Class> Bank
        <Property> accounts : Account [0..*]
        <Operation> depositOnAccount (id : st
        <Operation> withdrawFromAccount (i
      <Class> Client
    <Package> Common Java datatypes
```

```
Product7Bank.uml
platform:/resource/BankUML/Product7Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> limit : double
        <Operation> Account (i : string, m : d
        <Operation> deposit (amount : doubl
        <Operation> getAmount () : double
        <Operation> withdrawWithLimit (am
        <Operation> getLimit () : double
        <Operation> setLimit (l : double)
      <Class> Bank
        <Property> accounts : Account [0..*]
        <Property> cons : Consortium
        <Operation> Bank (c : Consortium)
        <Operation> depositOnAccount (id : :
        <Operation> withdrawFromAccount (
      <Class> Client
      <Class> Consortium
    <Package> Common Java datatypes
```

```
Product8Bank.uml
platform:/resource/BankUML/Product8Bank.uml
  <Model> Bank
    <Package> bs
      <Class> Account
        <Property> id : string
        <Property> amount : double
        <Property> currency : int
        <Operation> Account (i : string, m : do
        <Operation> deposit (amount : double
        <Operation> getAmount () : double
        <Operation> withdrawWithoutLimit (a
        <Operation> getCurrency () : int
        <Operation> setCurrency (c : int)
      <Class> Bank
        <Property> accounts : Account [0..*]
        <Property> converter [0..1]
        <Operation> Bank (c : Converter)
        <Operation> depositOnAccount (id : st
        <Operation> withdrawFromAccount (i
        <Operation> convert (curSource : int, c
      <Class> Client
      <Class> Converter
    <Package> Common Java datatypes
```

Fig. 1. Eight UML model variants for the banking systems

Model-Element(AME) represents a model element in the model variant. Then, we propose to reuse the algorithm proposed in [21] to identify commonality and differences in what is called features. The following sections presents SoAMEs and summarise the comparison algorithm.

Dividing a Model on a Set of Atomic Model-Elements: As shown by *Blanc et al.* [2] models are artifacts that can be expressed as a sequence of *elementary construction operations* and we have used a similar approach. By using the Meta Object Facility (MOF) concepts [15] we are able to divide any model compliant with the MOF specification. The AMEs in our approach are:

- *Class*: It consists of a "parent" Class and the class "object" itself.
- *Attribute*: It consists of an "owner" Class, an "attribute identifier" and a "value".
- *Reference*: It consists of an "owner" Class, a "reference identifier" and a set of "referenced" Classes.

We implemented this division of models using a pre-order tree traversal of the model by following the containment references. After adding each Class we add its Attributes and References. Before adding any of these AMEs we check that the structural feature (attribute or reference) is not derived, nor volatile, nor transient. If this is the case we are not interested in adding it to the SoAMEs. This check includes containment references as it could also happens. The reflexion capability of EMF models allows providing a generic approach for any meta-model to which MoVaC approach wants to be used.

By applying the presented method, Fig. 2 shows the division of the UML model corresponding to `Product1Bank` of the banking systems UML model variants. This figure shows only an excerpt of the 1049 AMEs of this model that contains 67 classes including UML classes, attributes, operations etc.

Model variants are thus represented as a SoAMEs. Formally, each model variant is defined as a set $M_i = \{ame_1, ame_2, ..ame_n\}$, where each $ame_i \in \{Class, Attribute, Reference\}$. In the following, we consider $AllMVs = \{M_1, M_2, ..M_N\}$ as the set of model variants that we want to compare.

Feature Identification Algorithm: Once we are able to divide models as SoAMEs, we reuse the algorithm proposed by *Ziadi et al.* [21] to calculate the commonality and variability between the model variants. This algorithm takes as input the SoAMEs of all model variants and identify differences and commonalities in the form of *features* where each feature is also a SoAMEs.

The feature identification process is based on a formal definition of a feature that uses the notion of interdependent AMEs. This notion is defined as follows.

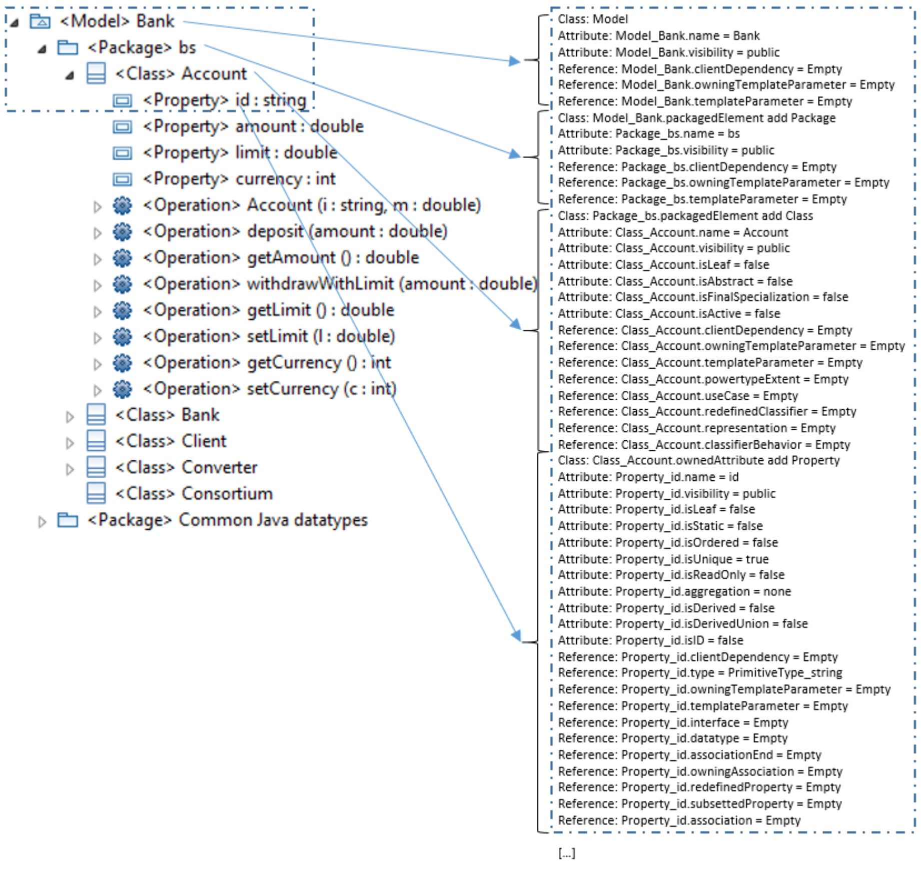
Definition 1 (Interdependent AMEs). *Given the set of model variants that we want to compare $AllMVs$, two AMEs (of models of $AllMVs$) ame_1 and ame_2 are interdependent if and only if they belong to exactly the same products of $AllMVs$. In other words, ame_1 and ame_2 are interdependent if the two following conditions are fulfilled.*

1. $\exists M \in AllMVs \;\; ame_1 \in M \wedge ame_2 \in M$.
2. $\forall M \in AllMVs \;\; ame_1 \in M \Leftrightarrow ame_2 \in M$.

Since interdependence is an equivalence relation on the set of AMEs of $AllMVs$, it leads us to the following definition of a feature.

Definition 2 (Feature). *Given $AllMVs$ a set of products, a feature of $AllMVs$ is an equivalence class of the interdependence relation of the AMEs of $AllMVs$.*

The application of the feature identification algorithm to the SoAMEs of the banking models provides the features depicted by Fig. 3. In order to ease the reading we only present representative AMEs from the full SoAME of each feature. The `Feature 0` gathers all the AMEs that are present in all the product variants. We have the `bs` package with the `Bank`, `Account` and `Client` classes as well as the shared package of the used data-types. The `Feature 1` concerns the limit information. This feature contains the Attribute AME related to the `withdrawWithoutLimit` operation in the `Account` class. A domain expert could



```
<Model> Bank
  <Package> bs
    <Class> Account
      <Property> id : string
      <Property> amount : double
      <Property> limit : double
      <Property> currency : int
      <Operation> Account (i : string, m : double)
      <Operation> deposit (amount : double)
      <Operation> getAmount () : double
      <Operation> withdrawWithLimit (amount : double)
      <Operation> getLimit () : double
      <Operation> setLimit (l : double)
      <Operation> getCurrency () : int
      <Operation> setCurrency (c : int)
    <Class> Bank
    <Class> Client
    <Class> Converter
    <Class> Consortium
  <Package> Common Java datatypes
```

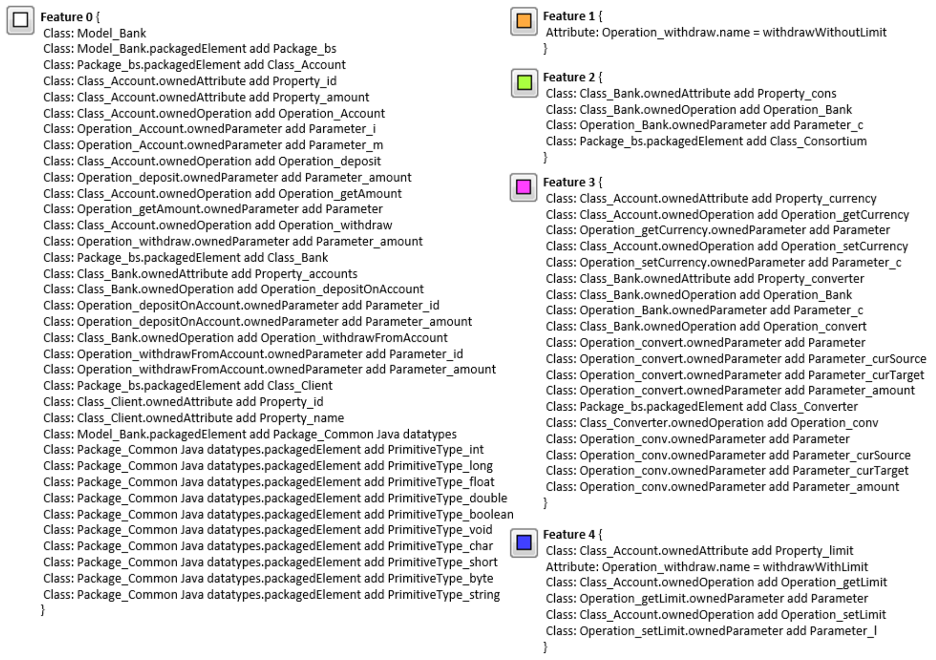
```
Class: Model
Attribute: Model_Bank.name = Bank
Attribute: Model_Bank.visibility = public
Reference: Model_Bank.clientDependency = Empty
Reference: Model_Bank.owningTemplateParameter = Empty
Reference: Model_Bank.templateParameter = Empty
Class: Model_Bank.packagedElement add Package
Attribute: Package_bs.name = bs
Attribute: Package_bs.visibility = public
Reference: Package_bs.clientDependency = Empty
Reference: Package_bs.owningTemplateParameter = Empty
Reference: Package_bs.templateParameter = Empty
Class: Package_bs.packagedElement add Class
Attribute: Class_Account.name = Account
Attribute: Class_Account.visibility = public
Attribute: Class_Account.isLeaf = false
Attribute: Class_Account.isAbstract = false
Attribute: Class_Account.isFinalSpecialization = false
Attribute: Class_Account.isActive = false
Reference: Class_Account.clientDependency = Empty
Reference: Class_Account.owningTemplateParameter = Empty
Reference: Class_Account.templateParameter = Empty
Reference: Class_Account.powertypeExtent = Empty
Reference: Class_Account.useCase = Empty
Reference: Class_Account.redefinedClassifier = Empty
Reference: Class_Account.representation = Empty
Reference: Class_Account.classifierBehavior = Empty
Class: Class_Account.ownedAttribute add Property
Attribute: Property_id.name = id
Attribute: Property_id.visibility = public
Attribute: Property_id.isLeaf = false
Attribute: Property_id.isStatic = false
Attribute: Property_id.isOrdered = false
Attribute: Property_id.isUnique = true
Attribute: Property_id.isReadOnly = false
Attribute: Property_id.aggregation = none
Attribute: Property_id.isDerived = false
Attribute: Property_id.isDerivedUnion = false
Attribute: Property_id.isID = false
Reference: Property_id.clientDependency = Empty
Reference: Property_id.type = PrimitiveType_string
Reference: Property_id.owningTemplateParameter = Empty
Reference: Property_id.templateParameter = Empty
Reference: Property_id.interface = Empty
Reference: Property_id.datatype = Empty
Reference: Property_id.associationEnd = Empty
Reference: Property_id.owningAssociation = Empty
Reference: Property_id.redefinedProperty = Empty
Reference: Property_id.subsettedProperty = Empty
Reference: Property_id.association = Empty

[...]
```

Fig. 2. Excerpt of a Bank UML model atomic elements

be able to analyse this and conclude that this is the case when the Bank has withdraw without limit. On the other hand, `Feature 4` presents the other possible case when the Bank has withdraw with limit. This way in `Feature 4` we have the Attribute AME related to the `withdrawWithLimit` operation. It also contains primitives to create the `limit` field, its getter and the method defining limit checking. `Feature 2` consists in the `Consortium` class and the given property and constructor operation of the `Bank` class. `Feature 3` includes all the needed classes and operations to manage currency exchange.

Comparing Atomic Model-Elements: The feature identification algorithm presented in previous section requires the definition of equals operator between AMEs. This section clarifies how this comparison between AMEs is performed. We rely mainly on existing techniques of model comparison that are highly extensible. These techniques enable to define when elements are equals or not



```
Feature 0 {
 Class: Model_Bank
 Class: Model_Bank.packagedElement add Package_bs
 Class: Package_bs.packagedElement add Class_Account
 Class: Class_Account.ownedAttribute add Property_id
 Class: Class_Account.ownedAttribute add Property_amount
 Class: Class_Account.ownedOperation add Operation_Account
 Class: Operation_Account.ownedParameter add Parameter_i
 Class: Operation_Account.ownedParameter add Parameter_m
 Class: Class_Account.ownedOperation add Operation_deposit
 Class: Operation_deposit.ownedParameter add Parameter_amount
 Class: Class_Account.ownedOperation add Operation_getAmount
 Class: Operation_getAmount.ownedParameter add Parameter
 Class: Class_Account.ownedOperation add Operation_withdraw
 Class: Operation_withdraw.ownedParameter add Parameter_amount
 Class: Package_bs.packagedElement add Class_Bank
 Class: Class_Bank.ownedAttribute add Property_accounts
 Class: Class_Bank.ownedOperation add Operation_depositOnAccount
 Class: Operation_depositOnAccount.ownedParameter add Parameter_id
 Class: Operation_depositOnAccount.ownedParameter add Parameter_amount
 Class: Class_Bank.ownedOperation add Operation_withdrawFromAccount
 Class: Operation_withdrawFromAccount.ownedParameter add Parameter_id
 Class: Operation_withdrawFromAccount.ownedParameter add Parameter_amount
 Class: Package_bs.packagedElement add Class_Client
 Class: Class_Client.ownedAttribute add Property_id
 Class: Class_Client.ownedAttribute add Property_name
 Class: Model_Bank.packagedElement add Package_Common Java datatypes
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_int
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_long
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_float
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_double
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_boolean
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_void
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_char
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_short
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_byte
 Class: Package_Common Java datatypes.packagedElement add PrimitiveType_string
}

Feature 1 {
 Attribute: Operation_withdraw.name = withdrawWithoutLimit
}

Feature 2 {
 Class: Class_Bank.ownedAttribute add Property_cons
 Class: Class_Bank.ownedOperation add Operation_Bank
 Class: Operation_Bank.ownedParameter add Parameter_c
 Class: Package_bs.packagedElement add Class_Consortium
}

Feature 3 {
 Class: Class_Account.ownedAttribute add Property_currency
 Class: Class_Account.ownedOperation add Operation_getCurrency
 Class: Operation_getCurrency.ownedParameter add Parameter
 Class: Class_Account.ownedOperation add Operation_setCurrency
 Class: Operation_setCurrency.ownedParameter add Parameter_c
 Class: Class_Bank.ownedAttribute add Property_converter
 Class: Class_Bank.ownedOperation add Operation_Bank
 Class: Operation_Bank.ownedParameter add Parameter_c
 Class: Class_Bank.ownedOperation add Operation_convert
 Class: Operation_convert.ownedParameter add Parameter
 Class: Operation_convert.ownedParameter add Parameter_curSource
 Class: Operation_convert.ownedParameter add Parameter_curTarget
 Class: Operation_convert.ownedParameter add Parameter_amount
 Class: Package_bs.packagedElement add Class_Converter
 Class: Class_Converter.ownedOperation add Operation_conv
 Class: Operation_conv.ownedParameter add Parameter
 Class: Operation_conv.ownedParameter add Parameter_curSource
 Class: Operation_conv.ownedParameter add Parameter_curTarget
 Class: Operation_conv.ownedParameter add Parameter_amount
}

Feature 4 {
 Class: Class_Account.ownedAttribute add Property_limit
 Attribute: Operation_withdraw.name = withdrawWithLimit
 Class: Class_Account.ownedOperation add Operation_getLimit
 Class: Operation_getLimit.ownedParameter add Parameter
 Class: Class_Account.ownedOperation add Operation_setLimit
 Class: Operation_setLimit.ownedParameter add Parameter_l
}
```

Fig. 3. Bank UML Models features

for a given specific purpose and to deal with meta-model peculiarities. MoVaC specifies a default comparison of AMEs that can be easily customized through extension points. More precisely, using the classification by *Kolovos et al.* [14] MoVaC default model comparison behaviour is static identity-based matching but MoVaC implementation provides standard Eclipse extension mechanisms to contribute signature based-matching if required.

We have implemented the equals boolean methods for each of the AMEs. We extensively used EMF DiffMerge that enables to compare two Model Scopes using Match and Diff policies.

- *Class*: Two *Class* AMEs are equals if we isolate each of the "objects" in a scope that contains only these elements and the Diff Policy returns no difference in the comparison. The Diff policy ignores all the attributes and references. This way it will not return that an `EObject` is different if they have different attributes or references. To check the "equals" for attributes and references we rely on the Attribute and References AMEs. The Match policy used by default consists in retrieving the ID attribute of the element and, if not defined, it tries to infer it checking different serialization mechanisms as XMI ids etc.
- *Attribute*: In EMF each defined attribute in the meta-model has an identifier (for example Operation_Name for the attribute Name of the Operation

meta-object). Two *Attribute* AMEs will be the same if they deal with the same attribute identifier and if the owner objects of the attribute are the same. Finally, the Diff policy is in charge of deciding whether the values of the attributes should be considered equals for this attribute identifier. The default implementation of the Diff policy just performs an equals operation on the values.

– *Reference*: As well as with Attributes, two *Reference* AMEs are equals if they share the same reference identifier and if the owner objects are the same. Then we check that the referenced classes are the same. If it is an ordered reference the objects must appear in the same positions. If not ordered then it is only needed that all the elements are present in the other Reference AME.

4.2 Step 2: Representation and Visualization

The visualization eases to show the different features and their presence and relative position in the set of model variants. By selecting the eight models that integrate the Banking systems UML Model variants we apply the MoVaC approach and we obtain the visualization presented on Fig. 4. Each bar represents one of the model variants and the stripes on each of the bars are the SoAMEs as computed by the division algorithm. This way the length of the bar represents the number of AMEs and consequently we will have bars with different lengths. As we can see the height of `Product1Bank` is greater than the height of `Product2Bank` as their SoAMEs sizes are 1049 and 567 respectively.

The MoVaC approach displays the list of identified features. Figure 4 shows the feature list previously mentioned for the Banking systems. Each feature has an assigned color. The stripes (AMEs) of the Model variants are colorized with the color of the feature that the AME belongs to. As illustrated in the Fig. 4, a specific feature could consist of AMEs that are scattered through the bar. This is because MoVaC displays AMEs of each model variant in the order that these AMEs are constructed in the first Calculation step. For example if we look at `Feature 2` of `Product5Bank` it has two parts. The first part will correspond to the Property and Operations related to Consortium in the Bank class while the second part will correspond to the Consortium UML class itself. The separation between the two parts are AMEs from other features. This helps locating and understanding the distribution of AMEs in model variants.

For the implementation of the visualisation we used the extensible visualiser of the Eclipse project AspectJ Development Tools [6]. It was originally used for visualising cross-cutting concerns in different modules. This visualisation method was already used for source code clone visualisation [20].

Apart from the main visualization presented here, other functionalities of this visualization are:

– *Filtering*: Using the checkboxes on the Feature list we can select which features we want to visualize. Also by selecting one of the bars we have the option to automatically show only the features that this bar contains.

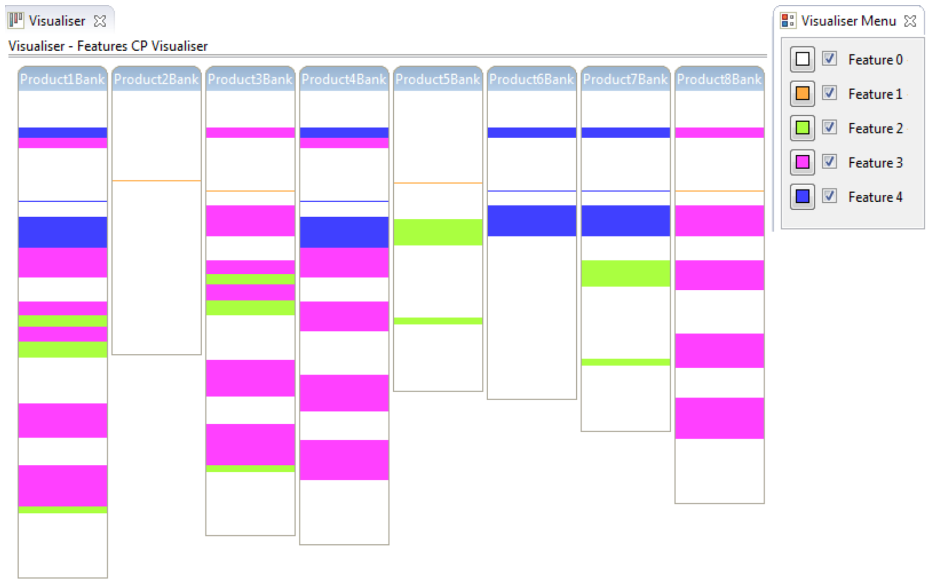


Fig. 4. Bank UML Model variants comparison

– *Analyze features*: The visualization has functionality to show the content of each Feature. That means that we can have the text representation of all the atomic elements that compose each of the Features.
– *Analyze atomic elements*: By clicking on a stripe of any bar we get the text representation of the selected atomic element.
– *Export*: It is possible to export the relation of Models and Features in a separated file that could be opened in a spreadsheet application for further processing or visualisation. Table 1 presents this relation between existing Banking systems and the Features.

Table 1. Relation of existing Bank models and identified features

	P1Bank	P2Bank	P3Bank	P4Bank	P5Bank	P6Bank	P7Bank	P8Bank
Feature 0	X	X	X	X	X	X	X	X
Feature 1		X	X		X			X
Feature 2	X		X		X		X	
Feature 3	X		X	X				X
Feature 4	X			X		X	X	

5 Evaluation and Discussion

5.1 Case Studies

In the last section, we applied our approach on the illustrative example concerning the Banking UML model variants. For the evaluation purposes, we present here a second case study. This example concerns Vending Machine variants represented as statecharts. These vending machines statechart variants were introduced in previous work [11]. The statecharts models are not UML based, the meta-model used is the Yakindu Statechart Tools meta-model [12]. Figure 5 shows the six analysed variants. The objective of these Vending Machines is to provide different kinds of drinks. For example, `VendingMachine1` provides only Soda while `VendingMachine4` provides all types of drinks. There could be also support for different payment methods for the customers. `VendingMachine1` only provides credit card payment while `VendingMachine1` only accepts cash. Some of the Vending machines, see `VendingMachine2` and `VendingMachine3`, alert the customer that the drink is ready through a ring tone. Apart from that, other states and transitions are common such as the `Idle`, `Select payment method`, `Deliver drink` and `Display message` states.

We applied the MoVaC approach to the six model variants for the Vending Machine. Figure 6 illustrates the obtained features. `Feature 0` gathers the SoAMEs shared by all the model variants. We have for example the main region and the mentioned states Idle, Product price displayed, Select payment method, Deliver drink and Display message. `Feature 3`, `Feature 4` and `Feature 5` contain respectively the SoAMEs related to the state transitions of entering the code of Soda, Coffee or Tea. `Feature 1` corresponds to the cash payment method while `Feature 6` corresponds to the credit card payment method. Finally, `Feature 2` contains the states and transitions regarding the ring tone alert.

The visualisation of the obtained features is presented on Fig. 7. Table 2 illustrates the relation between Features and model variants.

Table 2. Relation of existing Vending machine models and identified features

	SCT 1	SCT 2	SCT 3	SCT 4	SCT 5	SCT 6
Feature 0	X	X	X	X	X	X
Feature 1			X	X	X	X
Feature 2		X		X		X
Feature 3	X			X		X
Feature 4		X		X	X	
Feature 5		X	X	X		
Feature 6	X	X				

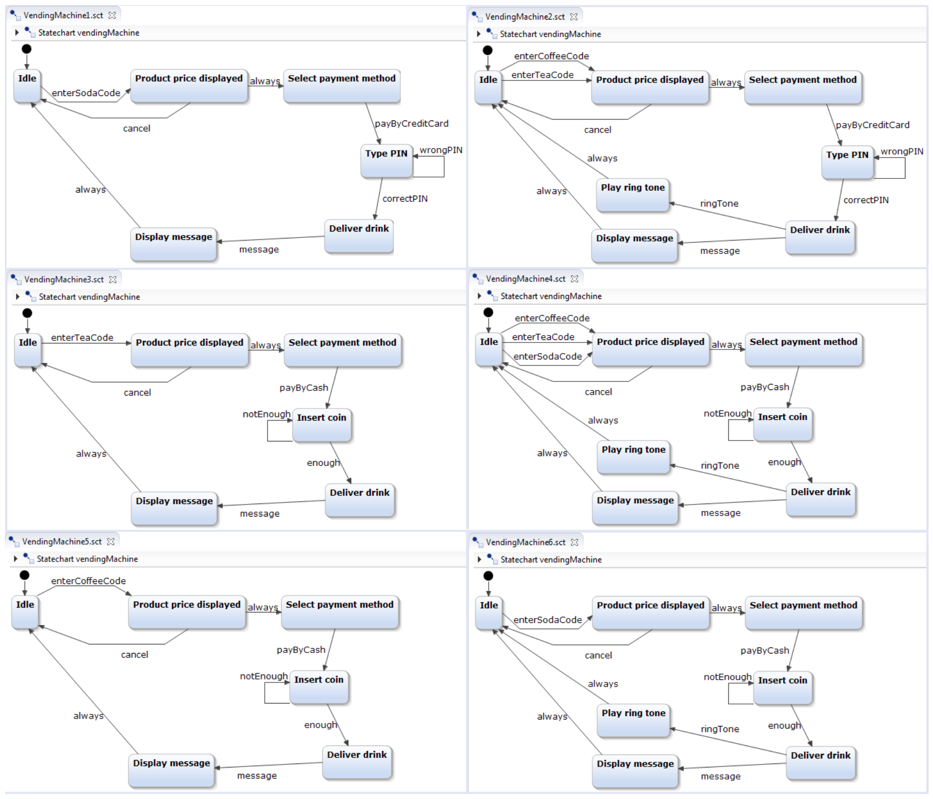


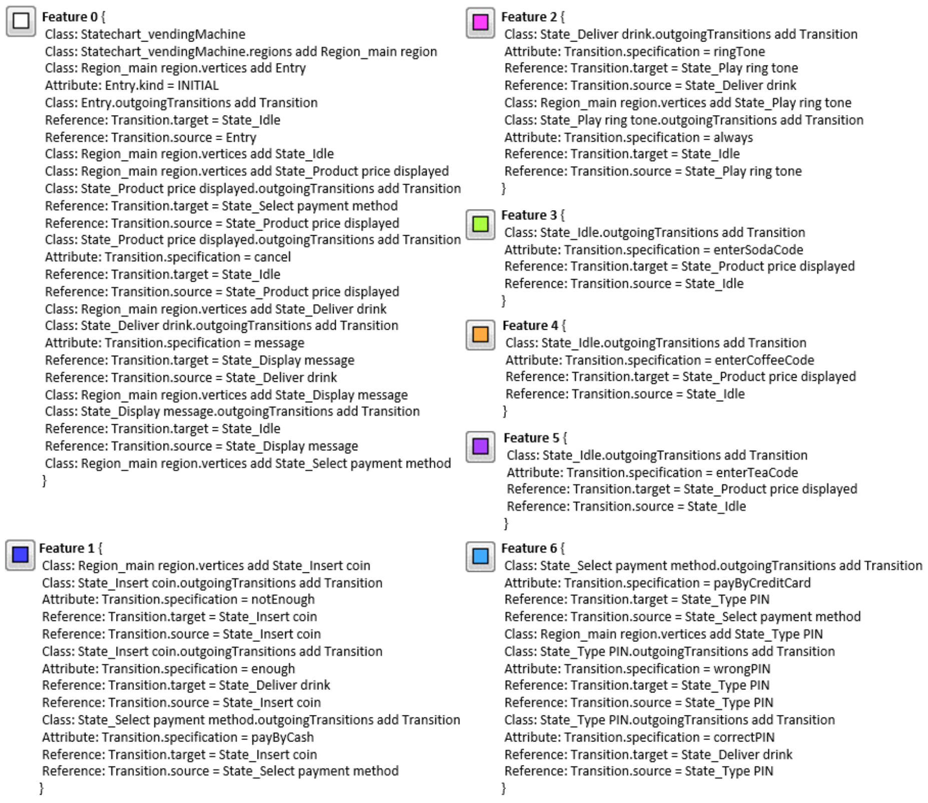
Fig. 5. Six SCT Vending machines model variants

5.2 Evaluation

In this section the MoVaC approach is assessed considering the following research questions:

- *RQ1: Soundness of the comparison.* Is the approach able to identify correctly the commonality and variability of a set of models?
- *RQ2: Flexibility.* Is the approach applicable to various modelling meta-models?

Soundness of the Comparison (RQ1) The first research question amounts to evaluate correctness of the commonality and variability identified and visualised with our approach. As mentioned above, the case studies that we used in this paper are inspired from papers [21](for the Banking UML model variants), and [11](for the vending machines). In addition to the model variants, these papers also present the actual commonality and variability in terms of features. We thus manually compared the obtained features with those initially



```
⬜ Feature 0 {
  Class: Statechart_vendingMachine
  Class: Statechart_vendingMachine.regions add Region_main region
  Class: Region_main region.vertices add Entry
  Attribute: Entry.kind = INITIAL
  Class: Entry.outgoingTransitions add Transition
  Reference: Transition.target = State_Idle
  Reference: Transition.source = Entry
  Class: Region_main region.vertices add State_Idle
  Class: Region_main region.vertices add State_Product price displayed
  Class: State_Product price displayed.outgoingTransitions add Transition
  Reference: Transition.target = State_Select payment method
  Reference: Transition.source = State_Product price displayed
  Class: State_Product price displayed.outgoingTransitions add Transition
  Attribute: Transition.specification = cancel
  Reference: Transition.target = State_Idle
  Reference: Transition.source = State_Product price displayed
  Class: Region_main region.vertices add State_Deliver drink
  Class: State_Deliver drink.outgoingTransitions add Transition
  Attribute: Transition.specification = message
  Reference: Transition.target = State_Display message
  Reference: Transition.source = State_Deliver drink
  Class: Region_main region.vertices add State_Display message
  Class: State_Display message.outgoingTransitions add Transition
  Reference: Transition.target = State_Idle
  Reference: Transition.source = State_Display message
  Class: Region_main region.vertices add State_Select payment method
}

🟦 Feature 1 {
  Class: Region_main region.vertices add State_Insert coin
  Class: State_Insert coin.outgoingTransitions add Transition
  Attribute: Transition.specification = notEnough
  Reference: Transition.target = State_Insert coin
  Reference: Transition.source = State_Insert coin
  Class: State_Insert coin.outgoingTransitions add Transition
  Attribute: Transition.specification = enough
  Reference: Transition.target = State_Deliver drink
  Reference: Transition.source = State_Insert coin
  Class: State_Select payment method.outgoingTransitions add Transition
  Attribute: Transition.specification = payByCash
  Reference: Transition.target = State_Insert coin
  Reference: Transition.source = State_Select payment method
}

🟪 Feature 2 {
  Class: State_Deliver drink.outgoingTransitions add Transition
  Attribute: Transition.specification = ringTone
  Reference: Transition.target = State_Play ring tone
  Reference: Transition.source = State_Deliver drink
  Class: Region_main region.vertices add State_Play ring tone
  Class: State_Play ring tone.outgoingTransitions add Transition
  Attribute: Transition.specification = always
  Reference: Transition.target = State_Idle
  Reference: Transition.source = State_Play ring tone
}

🟩 Feature 3 {
  Class: State_Idle.outgoingTransitions add Transition
  Attribute: Transition.specification = enterSodaCode
  Reference: Transition.target = State_Product price displayed
  Reference: Transition.source = State_Idle
}

🟧 Feature 4 {
  Class: State_Idle.outgoingTransitions add Transition
  Attribute: Transition.specification = enterCoffeeCode
  Reference: Transition.target = State_Product price displayed
  Reference: Transition.source = State_Idle
}

🟪 Feature 5 {
  Class: State_Idle.outgoingTransitions add Transition
  Attribute: Transition.specification = enterTeaCode
  Reference: Transition.target = State_Product price displayed
  Reference: Transition.source = State_Idle
}

🟦 Feature 6 {
  Class: State_Select payment method.outgoingTransitions add Transition
  Attribute: Transition.specification = payByCreditCard
  Reference: Transition.target = State_Type PIN
  Reference: Transition.source = State_Select payment method
  Class: Region_main region.vertices add State_Type PIN
  Class: State_Type PIN.outgoingTransitions add Transition
  Attribute: Transition.specification = wrongPIN
  Reference: Transition.target = State_Type PIN
  Reference: Transition.source = State_Type PIN
  Class: State_Type PIN.outgoingTransitions add Transition
  Attribute: Transition.specification = correctPIN
  Reference: Transition.target = State_Deliver drink
  Reference: Transition.source = State_Type PIN
}
```

Fig. 6. Vending Machine SCT Model variants features

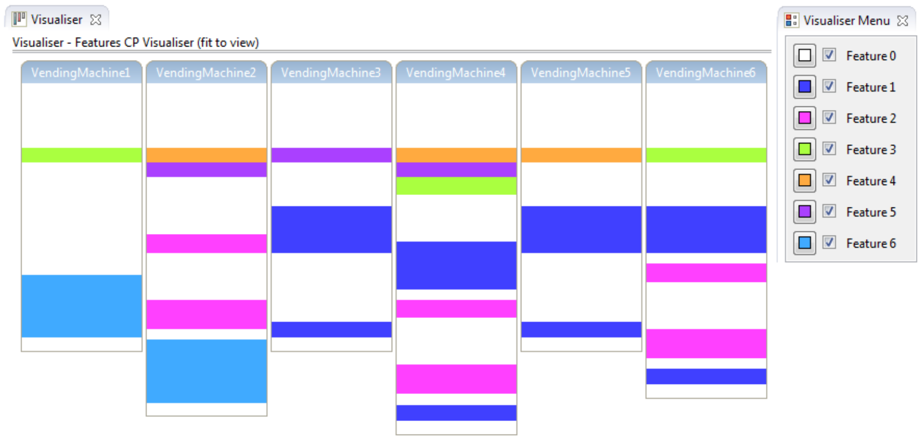


Fig. 7. Vending Machine SCT Model variants comparison

presented in these papers. The evaluation shows that that we get a full matching between commonality and variability identified by our approach and those initially presented in the mentioned papers.

Flexibility (RQ2) We validate our approach on two case studies that are based on two different meta-models. The MoVaC approach successfully presents to the user commonality and variability of model variants in terms of features. This shows that it is a generic approach and that it can be used for any EMF based meta-model. However, and as presented in Sect. 4.1, our approach depends on the AME's comparison method. In our implementation of the MoVaC approach we used EMF DiffMerge default policies but we are aware that current complex modeling tools require this AMEs comparison method to be extended to other policies. It is the case for SCT model variants that are based on the Yakindu Statechart Tool. We used MoVaC extensibility to implement two special cases to cope with the statechart tool's model serialization peculiarities. For instance, to improve performance, they include redundancy about the informations related to transitions in the serialized statecharts when it should be derived/calculated references. Also we found the usage of String Attributes as a mechanism to store text based domain specific languages. This ends up with issues while using the default EMF DiffMerge policies and this is way the extension mechanism of MoVaC was a requirement to cope with complex scenarios.

6 Conclusion

We presented the Model Variants Comparison approach as an enabler to identify and analyse commonality and variability in a set of models. As illustrated in this paper, MoVaC is a generic and customizable approach to analyse variability in space among different variants. MoVaC also implements the visualisation step where commonality and variability between model variants are presented to users in the form of what is referred to as features. We validated our approach on two case studies. As further work we aim to apply it in an industrial scenario dealing with huge model variants.

Acknowledgements

The present work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project MODEL C12/IS/3977071.

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML. Lecture Notes in Computer Science, vol. 2863, pp. 2–17. Springer (2003)

2. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE. pp. 511–520 (2008)
3. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. UPGRADE, The European Journal for the Informatics Professional 9(2), 29–34 (2008)
4. Eclipse: Eclipse modeling framework project (2014), http://www.eclipse.org/modeling/emf
5. Eclipse: Emf diff/merge: a diff/merge component for models (2014), http://eclipse.org/diffmerge/
6. Eclipse: The visualiser, ajdt: Aspectj development tools (2014), http://www.eclipse.org/ajdt/visualiser/
7. Engel, K.D., Paige, R.F., Kolovos, D.S.: Using a model merging language for reconciling model versions. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA. Lecture Notes in Computer Science, vol. 4066, pp. 143–157. Springer (2006)
8. Estublier, J.: Software configuration management: a roadmap. In: Finkelstein, A. (ed.) ICSE - Future of SE Track. pp. 279–289. ACM (2000)
9. Fenske, W., Thüm, T., Saake, G.: A taxonomy of software product line reengineering. In: Collet, P., Wasowski, A., Weyer, T. (eds.) VaMoS. p. 4. ACM (2014)
10. Girschick, M., Darmstadt, T.: Difference detection and visualization in uml class diagrams. Tech. rep. (2006)
11. Istoan, P., Biri, N., Klein, J.: Issues in model-driven behavioural product derivation. In: VaMoS. pp. 69–78 (2011)
12. itemis: Yakindu statechart tool (2014), http://statecharts.org
13. Kelter, U., Wehren, J., Niere, J.: A generic difference algorithm for uml models. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering. LNI, vol. 64, pp. 105–116. GI (2005)
14. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. pp. 1–6. CVSM '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/CVSM.2009.5071714
15. OMG: Meta object facility (mof) core specification (2006), http://www.omg.org/spec/MOF/2.0/
16. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques (2005)
17. Rubin, J., Chechik, M.: Combining related products into product lines. In: de Lara, J., Zisman, A. (eds.) FASE. Lecture Notes in Computer Science, vol. 7212, pp. 285–300. Springer (2012)
18. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Automatic variation-point identification in function-block-based models. In: GPCE. pp. 23–32 (2010)
19. Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Hammoudi, S., Pires, L.F., Filipe, J., das Neves, R.C. (eds.) MODELSWARD. pp. 265–277. SciTePress (2013)
20. Tairas, R., Gray, J., Baxter, I.: Visualization of clone detection results. In: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange. pp. 50–54. eclipse '06, ACM, New York, NY, USA (2006)
21. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. In: CSMR. pp. 417–422 (2012)