Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1007–1018
DOI: 10.15439/2015F364
ACSIS, Vol. 5

Feature Model Driven Generation of Software Artifacts

Roman Táborský and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava, Bratislava, Slovakia
E-mail: crudecrude@gmail.com, vranic@stuba.sk

***Abstract*—The objective of feature modeling is to foster software reuse by enabling to explicitly and abstractly express commonality and variability in the domain. Feature modeling alone is not sufficient to express all the aspects of the software being developed. Other models and, eventually, code is necessary. These software assets are being configured by the feature model based on the selection of variable features. However, selecting a feature is far from a naive component based approach where feature inclusion would simply mean including the corresponding component. More often than not, feature inclusion affects several places in models or code to be configured requiring their nontrivial adaptation. Feature inclusion recalls *transformation* and this is at heart of the approach to feature model driven generation of software artifacts proposed in this paper. Features are viewed as solution space transformations that may be executed during the generative process conducted by the feature model configuration.**

***Index Terms*—feature modeling; transformation; metatransformation; generative process; reuse**

I. Introduction

FEATURE modeling is an approach used in software development proposed in 1990's [1] with the growing popularity of software product lines. The objective of this approach to modeling is to foster software reuse by enabling to explicitly and abstractly express commonality and variability in the domain. Based on commonality and variability, appropriate implementation mechanisms can be selected [2].

Feature modeling alone is not sufficient to express all the aspects of the software being developed. Other models and, eventually, code is necessary. These software assets are being configured by the feature model based on the selection of variable features. However, selecting a feature is far from a naive component based approach where feature inclusion would simply mean including the corresponding component. More often than not, feature inclusion affects several places in models or code to be configured requiring their nontrivial adaptation. Feature inclusion recalls *transformation* and this is at heart of the approach to feature model driven generation of software artifacts proposed in this paper.

The rest of the paper is organized as follows. Section II discusses the possibilities of representing software artifacts in feature modeling. Section III identifies specifics of feature modeling necessary for the employment of this technique in the generative process. Section IV explains how features can be perceived as transformations, which is the essence of the approach proposed in this paper. Section V presents the overall process of feature model driven generation of software artifacts. Section VI reports the implementation of the approach. Section VII presents the evaluation. Section VIII discusses related work. Section IX concludes the paper.

II. Representing Software Artifacts in Feature Modeling

Feature modeling can be used to configure software assets—models and code—in order to create software instances that exhibit desired features. One way to achieve this is by employing so-called superimposed variants [3] where the software models or code contain all the variants that are being reduced based on the features selected—or not selected—in the corresponding feature model [4], [5]. The FeatureHouse project [6] implements an approach that uses these models and allows language independent source code generation.

The pure::variants software tool [7] uses a specialized family model to represent a feature to architecture mapping. In this model, it is necessary to specify the type of the impact on the software instance. There are several possible impact specifications that allow for a wide scope of software artifact to be created, such as files, file fragments, XSLT transformations, conditional XML or text, C/C++ flag files, makefiles, class alias files, or symbolic links to folders or files.

These types of interaction can be effectively used to describe various architectural parts of elements, but provide no way of direct implementation of quality attributes [8]. Extra-functional[1] features can be mapped to specific modules/components or software artifacts in general by implying rules the software artifacts have to comply with, such as specific testing procedures or documentation requirements, or by specifying a human interaction task to be performed, such as evaluating a managerial decision or performing human assessment of a feature instance in a finalized product (e.g., evaluating user experience or GUI usability) [10].

In automatic software composition, software artifacts represent the reusable and generated parts of software. By putting together these parts and rules expressed by extra-functional

[1]We use term *extra-functional*—as proposed by Mary Shaw [9]—to refer to requirements and features that go beyond software system functionality instead of more widely used, but potentially confusing term non-functional.

978-83-60810-66-8/$25.00©2015, IEEE

features, it is possible to automatically compose large parts of software.

III. Feature Modeling for the Generative Process

To drive the generative programming by a feature model, it is necessary to represent the software assets in a configurable manner. Also, the feature model must be capable of expressing different transformations of the software assets to produce the software artifacts that correspond to the features that have been selected.

A. Feature Types

As has been explained in Section II, features can be functional or extra-functional. Some functional features exhibit a crosscutting nature, i.e., they affect several distinct and often unrelated parts of the system. Extra-functional features commonly correspond to the quality requirements (e.g., security or performance) or to the impact of the software being created to its environment.

Furthermore, internal and external features can be distinguished. Internal features are contained within the software being created and they can be changed during the development process. External features represent environment functionality and quality implications and these are part of the deployment environment and thus are not affected by the development process. An example of an internal feature is a configuration file for a web server to which the product is going to be deployed. The web server and its running environment is an example of an external feature.

B. Feature Implications

The inclusion or exclusion of specific features in a particular feature model configuration has an impact on the final software product being created. Apart from the features that serve purely organizational purpose, which is mostly to group other features, the features in a feature model can be viewed as abstract elements having some impact on the final software product. This idea is crucial to automated software generation and configuration because a feature can be viewed as prescribing a change to the configuration or generative process. This means that the whole generative process consists of a set of events that are implied by the inclusion or exclusion of the features in a specific configuration.

The set of features that are included in the generative process is a result of a feature model configuration process. This process resolves variability in the feature model and a fully-specified feature model configuration can be used as an input to the generative process.

A feature inclusion can have various kinds of impact on a final software product: file or folder operations, deployment rules to provide documentation, realizing the testing requirements, etc. It is necessary to distinguish between a fully autonomous feature impact that requires no human interaction to deliver a final software asset (e.g., source code generation) and an impact that requires human interaction (e.g., designing a splash screen). With respect to the fully automated generative programming approach, the latter is not applicable. One way of integrating human interaction into the generative process is to create task placeholders that notify developers their input is needed for the process to continue.

Autonomous actions can be described by a set of events that perform a particular computer operation. This leads to the concept of extending the feature model by including the information of these events that are implied by the specific features in the model. By this, we get the description of the generative process event chain that has to be executed when processing a particular configuration of a feature model.

C. Generative Process

A generative process based on a feature model is driven by its configuration, i.e., by selecting the features. This process represents the transformation of the input models or code into resulting software artifacts according to the feature selection. The process can be restricted to only a single solution space transformation or it can represent a complex multi-tier set of intermediate actions where each of these can be perceived as a stand-alone process with its own inputs and outputs. In any case, the key issue is how to realize the impact of the feature inclusion on the underlying software assets. This is different for functional and extra-functional features.

1) Functional Feature Inclusion: Functional features can be directly mapped to software artifacts such as source code or resource files. The relationships between functional features can be quite complex and the problem of feature interaction can aries.

A simple example of a functional feature is the choice of the data provider for some other feature. The corresponding feature diagram is displayed in Figure 1. In this paper, a simple FODA-style feature diagram notation [1] is used. A feature diagram is a tree whose nodes represent features that can be selected or not for the resulting configuration. For a feature to be selected, its parent feature has to be selected. Empty circle ended edges connect parent features with their optional features. An arc over a group of edges means the corresponding features are mutually exclusive (alternative). Non-decorated edges connect mandatory features. We do not elaborate on textually expressed constraints and default dependency rules that are necessary to overcome the limitations of the tree structure of feature diagrams [11], [12], as for the purposes of the approach proposed here, these can be considered as any other feature relationships to be applied in feature model configuration.

A configurable piece of code corresponding to this model could in C# look like this:[2]

```
DataProvider provider = new DataProvider();
provider.DataSource = new <Template Field>();
Page.GridView.DataSource = provider;
Page.Databind();
```

Choosing the *XML Data source* changes the second line of the code sample to the following one:

[2]The further examples in this paper are in C#, too, if not stated otherwise.

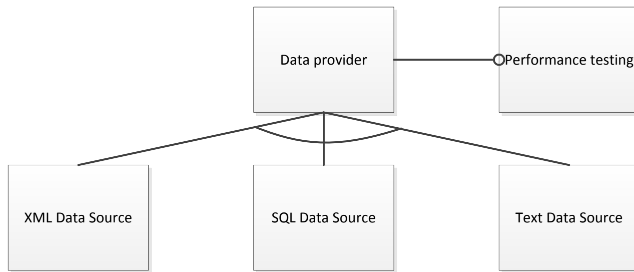

Fig. 1. The choice of the data provider as a functional feature and performance testing as an extra-functional feature.

```
provider.DataSource = new XMLDataSource(params []);
```

Another example would be an inclusion of the logging feature. This example is a little bit more complex as the points in source code to which the feature has to be bound are changing with as the code base grows. Aspect-oriented programming can be applied here to express so-called join points declaratively and address them without having to actually modify their source code representation.

2) Extra-Functional Feature Inclusion: Extra-functional features are a more difficult problem than functional features as it is rarely possible to map them to functional software artifacts. However, it is possible to implement the tests of the corresponding software artifacts against the conditions stated by extra-functional features. Consider the performance testing requirement as an example. The corresponding optional performance testing feature (see Figure 1) represents the fact that the software product has to conform to performance criterion and therefore needs to be tested against this criterion. This type of a feature can be directly represented in source code:

```
class XMLDataSource {
}
```

Having the *XMLDataSource* class to represent a functional feature when the extra-functional feature *Performance testing* is included, the source code can be transformed as follows:

```
class XMLDataSource {
    private TestContext testContextInstance;
    ///<summary>
    ///Gets or sets the test context which provides
    ///information about and functionality for the current test run.
    ///</summary>
    public TestContext TestContext {
        get {
            return testContextInstance;
        }
        set {
            testContextInstance = value;
        }
    }
    [TestMethod]
    public void PerformanceTesting() {
        throw new NotImplementedException();
    }
}
```

The *XMLDataSource* class has been enriched by methods and properties that support the performance testing. Among these is the *PerformanceTesting()* method, which has a placeholder that states it must be implemented manually.

IV. Features as Transformations

Fully configured feature model provides a list of all features that are affecting the generative process. Each of these features that are included provides a partial information on what actions shall be taken during the automated software creation. A transformation is an entity that represents these actions to be taken for a particular feature. Consequently, the generative process becomes a composition of all transformations included in the particular configuration that was the input for generative process. A problem with this approach is that it is necessary to define the order in which the transformations will be executed. There are several possible solutions:

- Explicitly adding the ordering information in the input feature model configuration
- Traversing the feature model configuration structure in a predefined way
- Providing a priority property to transformations

An explicitly stated order in the feature model depends on the knowledge of priority in which the transformations have to be executed. This means that if we add a new feature into the model, the whole model needs to be examined to accommodate the changes in priority. Traversing the feature model in a predefined way is based on an idea that a feature model is a tree, so it can be scanned breadth-first or depth-first. In both cases, a priority parameter has to be introduced into features.

Many actions are common to different features. Consider creating a file, creating a folder, or making a text input text into a file. These transformations are actually generic, but rely on specific parameters in achieving their result. Therefore, a transformation is in fact a transformation template that uses the information provided to create a specific transformation instance. Consequently, a transformation in our approach is an entity consisting of the following elements:

- List of events, i.e., actions to be performed
- List of requirements that are imposed on the feature model
- List of metatransformations that are included in this transformation

Transformations are bound one-to-one to features in the feature model. This means that every feature that is included in the feature model and is relevant to the generative process of the final solution has its a transformation assigned to it. The feature node in the feature model also carries all the additional information that is bound to the transformation assigned to it.

A. Transformation Reusability

The process of the transformation design requires an interaction on part of a domain engineer to provide the necessary domain information specific to the project and a software

engineer to design the transformations in such a way that they implement the information provided by the domain engineer and the requirement analysis of the corresponding features. For an effective cooperation between the domain and software engineer, it is useful to distinguish different levels of transformations with respect to reuse:

- Specialized transformations that can be used only for the specific features in the specific configuration of the feature model
- Specialized transformations that can be used only for the specific features, but in any configuration of the feature model
- Domain dependent generic transformations, which can be used across multiple software product lines in the same domain
- Domain independent generic transformations, which are the most reusable transformations as they can be included in different software product lines across multiple domains

Distinguishing these transformation types helps designing transformations that are as generic as possible at their level.

B. Transformation Hierarchy

An atomic transformation is a transformation impossible or unwanted to be decomposed into smaller transformations. Thus, even though such a transformation may consist of differentiable actions, the transformation is conceptually perceived as a one. Complex transformations are transformations that can be decomposed into a set of transformations where each of these lower level transformations represent an atomic transformation or a complex one (see Figure 2). Complex transformations themselves can have their own event chain besides the event chains of atomic transformations they embrace.

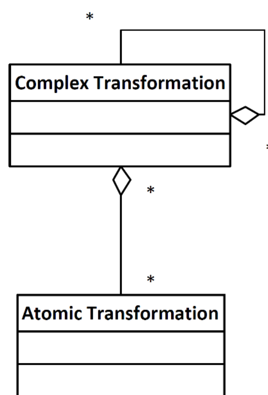

Fig. 2. Atomic and complex transformations (UML).

C. Transformation Inheritance

The inheritance relationship known from object-oriented programming can be applied to transformations, too (see Figure 3). It is possible to use inheritance as a way of combining ancestor and descendant event chains. There are several possible approaches to inheriting event chains:

- The ancestor event chain remains the same
- Items are added or removed to the event chain
- The event chain is completely overridden in the descendant transformation

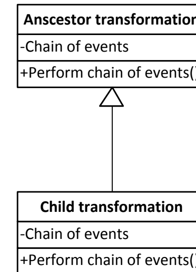

Fig. 3. Inheritance between transformations (UML).

The inheritance model can also be perceived as a way of organizing transformations into logical groups or packages. With respect to this, the inheritance is purely a tool of categorization and it is not necessary to maintain the typical parent–child class relationship, i.e., if transformations are represented as classes, it is not necessary to support inheritance mechanism at the level of methods and attributes.

D. Metatransformations

There are some features that have a global effect that spans throughout the whole software product (or its significant part). Quality features, such as logging requirement or performance and security constraints, represent a typical example. Including such a feature with the corresponding transformation into the configuration leads to the necessity of modifying other transformations in one of the following ways:

- Modifying the event chain of a transformation
- Modifying the requirements of a transformation
- Providing information that is defined by the transformation requirements

The modification of the event chain means that some actions are added or removed to or from the transformation being modified, or some properties of these actions are changed. This leads to the connection with the other two types of modification that can be defined on their own or are a result of this first type of modification.

E. Transformation Requirements

Seeing a transformation as a transformation template that instantiates a specific transformation based on external information, such as the file name in the create file transformation, constitutes the need for the external information required to perform this transformation. Therefore, it is necessary to include the information in the feature model or the associated feature model configuration to allow for this. However, having two features represented by the same type of transformation, the information provided can vary (see Figure 4).

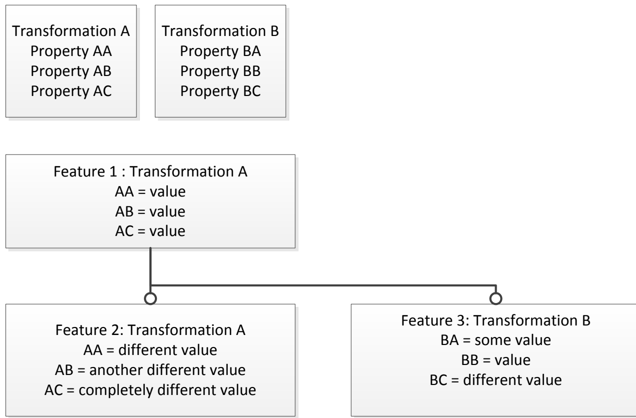

Fig. 4. Mapping transformations and properties to features.

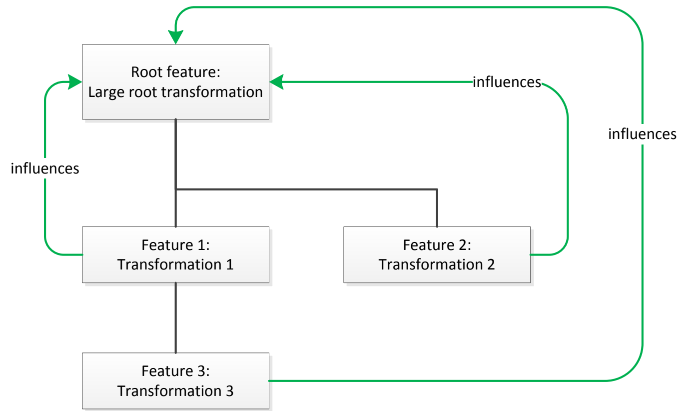

Fig. 6. The transformations influencing a large root transformation.

Each transformation defines these requirements as a list of items that contain the particular facts about the requirements. This means that upon including a transformation in the feature model, it is necessary to specify its requirements.

F. Transformation Granularity

There are two extreme approaches with respect to extending feature models with transformations. One approach is to target the separation of concerns with a large number of small transformations. This allows us to isolate the implications of transformations on the final software product into many small groups independent of each other (see Figure 5). The advantage of this approach is that the transformation model is easily changed without the need to analyze the impact on the whole transformation/feature model tree.

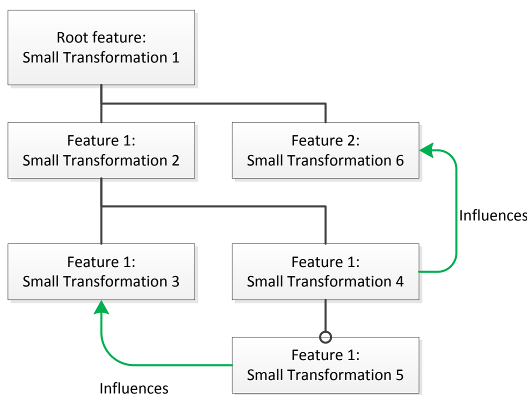

Fig. 5. Relationships between small independent transformations.

The other approach is to describe most actions in one global transformation that can be assigned to the root feature in the feature model. This global transformation is afterwards modified by the metatransformations that are assigned to the rest of the feature model nodes and their execution is based on the inclusion or exclusion of these features in a specific configuration (see Figure 6).

With respect to the transformation granularity, it is also important to consider the number of complex transformations in the model. Again, there are several approaches that can be used. One approach is to define only the simplest actions as atomic transformations and then to compose other transformations out of these atomic transformations. Another approach is to provide fairly large transformations that perform a substantial part of the tasks connected to a particular feature. It is also possible to employ an approach that lies somewhere inbetween of these extreme viewpoints. This is possible for the high or low granularity extremes and also for the complexity aspect of transformations. For example, metatransformations can be avoided by providing specific transformations for each feature.

G. Including Transformations in the Feature Model

Since one transformation corresponds to one feature, their inclusion into the feature model means simply assigning the corresponding information to each feature (see Figure 7).

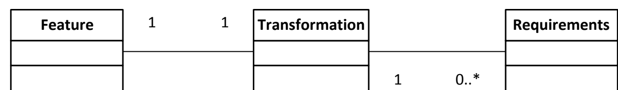

Fig. 7. Transformations are associated with features (UML).

One way of connecting the feature model with transformations is to include it directly in the feature model (an XML representation in used in this example):

```
<feature>
    <feature>
        <transformation>
            <!−− Transformation information including
            the event chain, metatransformation
            information, requirements, etc. −−>
        </transformation>
    </feature>
</feature>
```

There are two main problems connected with this approach: the degree of the transformation reuse between different models is reduced and it is necessary to parse the transformation

information separately for every feature, even though the type of the transformation they use can be the same (e.g., create a file).

Another approach is to store the transformation definitions outside the feature model. A sample structure of this can be a feature model represented in XML and the transformations defined as C# classes that are used by the generator:

```
<feature transformationClassName="CreateFile"
   fileName="Samplefile.cs">
</feature>

public class CreateFile : Transformation {
   public override void ExecuteTransformation() {
      ... // Create a file
   }
}
```

With this approach, it is necessary only to specify the transformation type and requirements of this transformation. The information about the metatransformations is not included in the feature model as it is internal to the transformation system and including this information in the feature model would be redundant.

V. The Overall Process

The overall process of employing the feature model driven generation of software artifacts is as follows:

1) The input to the process is a feature model
2) Transformations are assigned to the feature model
3) Transformation requirements are provided where possible
4) A specific configuration of the feature model is created
5) Configuration specific requirements of the transformations are provided
6) The metatransformations are executed
7) The requirements and changes to the configuration model are incorporated or provided
8) The generator processes the fully specified configuration of the feature model and performs the event chain of the transformations included in it
9) The output of this process is a generated instance of the software system based on the input feature model, its configuration, and the transformations specified in the feature model

Transformation requirements are being fulfilled at three levels:

1) Feature model level
2) Feature model configuration level
3) Feature model configuration level after the execution of metatransformations

One may wonder whether it is not possible to merge the latter two levels into one. This is not possible because the metatransformations depend on the information provided at level 2 and therefore it is necessary to provide this information before executing level 3.

Often, the impact of a feature inclusion is not easy to analyze. Therefore, a stepwise approach to the transformation defining the impact of a feature design can help to analyze the impact of each atomic operation that a transformation consists of and also it can help to decide the order in which the transformations are applied. This is important because in many cases the final impact of a transformation depends on the order of the transformation in the sequence in which transformations are executed.

In the manual transformation application, it is not necessary to have this order predefined, but when a software asset generator employs an extended feature model as an input to perform the execution sequence, it must be deterministic and specified before the generative process starts as the generator is not aware of the final implications during the generative process: it merely executes the transformations that result from the input feature model configuration.

In summary, the main objectives of the stepwise transformation design are:

- Create complex transformation by a stepwise application of low-level transformations
- Assess the solution space after each low-level transformation in a stepwise manner
- Analyze the whole generative process by exploring it by this stepwise approach

When applying this approach, it is necessary to record the steps taken in the manual application of transformations. This recording can be further specified to create a complex transformation that can be used in the generative process. It also allows to analyze the impact of transformations that perform mass file renaming or other large-scale operations. The last important information contained in this recording is the transformation order, which allows to analyze and prioritize transformations in the generative process.

VI. Implementing the Transformation Approach

The proposed approach of the feature model driven generation of software artifacts has been implemented in the .NET framework. A study of implementing the family of simple web sites has been performed.

A. Applying the Transformation

It takes three steps to apply a transformation:

- Process all metatransformations in the model
- Check for requirements
- Execute the transformation

Consider the transformation called *CreateStaticHTMLPage* as an example. This transformation utilizes its parameters to fill a predefined HTML template string that is processed with the *String.Replace()* method. The template defines the HTML file and contains placeholders that are replaced with the values of the transformation parameters:

```
#region html text template
protected string htmlContent =
   "<!DOCTYPE html PUBLIC \"−//W3C//DTD XHTML 1.0
      Strict//EN\"
   \"http://www.w3.org/TR/xhtml1/DTD/xhtml1−strict.dtd\">" +
   "<html xmlns=\"http://www.w3.org/1999/xhtml\"
```

```
        xml:lang=\”en\”> <head> ” +
    ”<meta http−equiv=\”content−language\”
        content=\”en\” />” +
    ”” +
    ”<title>Title placeholder</title>” +
    ”</head>” +
    ”” +
    ”<body>” +
    ”Content placeholder” +
    ”</body>” +
    ”</html>”;
#endregion
```

This transformation has four parameters:

- *PageName*: the file name that is used when creating the HTML file
- *htmlTemplate*: the template that is set up can be customized with this parameter
- *htmlTitle*: the text that replaces the *Title* placeholder
- *htmlContent*: the text that replaces the *Content* placeholder

If the transformation is included in the feature model, the configuration with these parameters:

```
<feature name=”StaticContent−History” ID=”2”
    Transformation=”org.crd.dp.CaseStudy.SimpleWebFinal,
        org.crd.dp.CaseStudy.
    SimpleWebFinal.Transformations.CreateStaticHTMLPage”
    htmlContent=”&lt;h1&gt;History&lt;/h1&gt;&lt;p&gt;
        Lorem Ipsum&lt;/p&gt;”
    htmlTitle=”StaticPage− History”
        PageName=”Site\\History.html” />
```

creates the following HTML file:

```
<!DOCTYPE html PUBLIC ”−//W3C//DTD XHTML 1.0
    Strict//EN”
    ”http://www.w3.org/TR/xhtml1/DTD/xhtml1−strict.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml” xml:lang=”en”>
    <head>
        <meta http−equiv=”Content−Type”
            content=”text/html; charset=utf−8” />
        <meta http−equiv=”content−language” content=”en” />
        <title>StaticPage− History
        </title>
    </head>
    <body><h1>History</h1>
        <p>Lorem Ipsum
        </p>
    </body>
</html>
```

B. Feature Model Configuration

In this sample implementation, the feature model configuration is represented by an XML file. The structure of this file follows the tree structure of the feature diagram:

```
<?xml version=”1.0” encoding=”utf−8” ?>
<featuremodel>
<feature>
    <feature>
        <feature />
        <feature />
    </feature>
    <feature />
< /feature>
</featuremodel>
```

Each XML feature node has three compulsory attributes:

- name
- ID
- Transformation

Accordingly, the simplest feature node looks like this:

```
<feature name=”DynamicContentProvider” ID=”7”
    Transformation=”Transformations.Empty” />
```

The *name* and *ID* attributes have solely the purpose of identifying the node when the node is processed. The *Transformation* attribute specifies the transformation that will be used with this feature.

The transformation attribute consists of two parts delimited by a comma. The first part represents the dynamic link library that contains the transformation, and the second part the full class name of the transformation. The dynamic link library has to be a .NET managed library. Therefore, a filled transformation attribute looks like this:

```
Transformation=”org.crd.dp.CaseStudy.SimpleWebFinal, ...
    Transformations.Empty”
```

This model is afterwards transformed into an object model contained within the *TransformationStore* object.

C. Transformations

The transformations in this implementation are based on a single class called *Transformation*. This class implements the *ITransformation* interface. This interface provides the basic methods needed by the generator to process the transformation:

```
public interface ITransformation {
    CheckPrerequisites();
    ExecuteTransformation();
    GetParameterNames();
    GetMetaTransformations();
    ...
}
```

The *CheckPrerequisites()* method does basic requirement checking before a transformation is processed. In this implementation, only a basic verification whether the transformation requirements are fulfilled is performed. However, it is possible to write a model aware method at the level of a metatransformation that can check the transformation dependencies. A sample of a model aware transformation is provided with the dynamic access log page.

The *ExecuteTransformation()* method represents the action which is contained within the transformation. This method is called in the final step of model processing.

The *GetParameterNames()* method is used in the XML file parsing, when it provides the parameter names to be retrieved from the feature node attributes.

The *GetMetaTransformations()* method provides a way how to retrieve the metatransformations that are connected with this transformation. This allows to connect a metatransformation list with a transformation and by this provide it with model awareness, which means that it can influence other transformations in the model within the possibilities provided by the connected metatransformations.

Two lists are initialized in the transformation class constructor:

```
protected Transformation() {
    metaTransformations = new List<MetaTransformation>();
    parameters = new Dictionary<string, object>();
}
```

These two lists contain the metatransformation list and key–value pairs of parameters. As the metatransformation list is of type *List <MetaTransformation>* and the interface requires *List <IMetaTransformation>*, it has to be casted:

```
public virtual List<IMetaTransformation>
    GetMetaTransformations() {
    return metaTransformations.ConvertAll(
        mt => (mt as IMetaTransformation));
}
```

The implementation of the transformation class provides two other important methods, *SetParameter()* and *GetParameter()*, which are the basis for parameter provisioning:

```
public object GetParameter(string name) {
    return parameters[name];
}
public void SetParameter(string name, object value) {
    if(parameters.ContainsKey(name))
        parameters[name] = value;
    else
        parameters.Add(name,value);
}
```

D. Composite transformations

To cope with complex transformations, composite transformations, i.e., transformations that contain other transformations can be used. In the following implementation sample, the composition is based on overridden methods in a descendant class:

```
public class CompositeTransformation : Transformation {
    private TransformationStore transformations;
    protected CompositeTransformation(): base() {
        transformations = new TransformationStore();
    }
    public override string CheckPrerequisites() {
        foreach(ITransformation transformation
            in transformations.Store)
        { transformation.CheckPrerequisites();
        }
        ...
    }
    public override void ExecuteTransformation() {
        foreach(ITransformation transformation
            in transformations.Store)
        { transformation.ExecuteTransformation();
        }
    }
    ...
}
```

As it is observable form the sample code, the composite transformation contains *TransformationStore*, which is basically an ordered list of transformations. The *ExecuteTransformation()* and *CheckPrerequisites()* methods operate on this list. In the case this list contains another composite transformation, the transformations are processed in a depth-first recursive way.

The problem that arose with implementing this list was that at the point of creating an instance of a composite transformation, the child transformations could not access the parent transformation parameters as these were not yet extracted from the XML file and often these child transformations rely on the information provided in the model. Therefore, a new method called *InstantiateChain()* was introduced to the composite transformation. This method is called after the transformation requirements are extracted to the parent transformation. The name of this method suggests that this child transformation list represents a list of events as described in Section IV, which can be modified by metatransformations as suggested in Section IV-D.

E. Metatransformations

The metatransformations are implemented by the *MetaTransformation* class. The difference between this class and the *Transformation* class is that the *MetaTransformation* class is model aware. This means that it can traverse the feature model and make changes to it. The specific changes are shown with transformation implementation samples. The difference is in the *ExecuteTransformation method()*:

```
ExecuteTransformation(object model, TransformationStore store);
```

In this method, the *TransformationStore* object is passed, representing the feature model. To allow for the modification of composite transformations, this method is implemented in a recursive way:

```
ExecuteTransformation(objectmodel, TransformationStorestore) {
    ...
    foreach(var trans in store.Store) {
        if(IsSubclassOfClass(typeof(
            CompositeTransformation),trans.GetType())) {
            ExecuteTransformation(model,
                ((CompositeTransformation)trans).Store);
        }
    }
}
```

F. Generator

The generative process consists of these steps:

- Parse the feature model configuration from the XML file

- Parse and execute the metatransformations contained within the transformation from the XML file
- Check the requirements of the parsed transformations
- Execute the transformation

The step of checking the requirements before parsing the metatransformation is omitted as it is contained within the metatransformation parsing step. To support these steps, a parser object is introduced. This object is represented by the *IFeatureModelParser* interface:

```
public interface IFeatureModelParser {
    ParseFeatureModel(...);
    ExecuteTransformationChain(...);
    ParseMetaTransformations(...);
    CheckPrerequisites();
}
```

This object contains the methods that provide the functionality required to perform these steps. The *XMLModelParser* class is used to represent this object. This class provides the functionality required based on an XML feature model representation. The steps will be described in separate sections.

1) Parse the Feature Model Configuration File: The first step that is necessary is to translate the transformations from the XML feature model configuration into an object model. The transformations are parsed in a top-down order. It is possible to override this behavior using the *Priority* attribute at transformation nodes. First, the assembly and class name are parsed from the XML node and basic reflection is performed to create an instance of the transformation:

```
var assembly = Assembly.Load(assemblyName);
var ttype = assembly.GetType(typeName);
var transformationInstance = ttype.GetConstructor(
    Type.EmptyTypes).Invoke(null) as ITransformation;
transformationInstance.SetID(Convert.ToInt32((string)
    node.Attributes["ID"].Value));
```

After creating an instance of the specified transformation, the *GetParameterNames()* method is used to obtain the list of parameters that this transformation uses. Afterwards, the XML node attributes that correspond to this list are copied into the dictionary containing the parameter key–value pairs:

```
foreach (var parameter in parameterNames) {
    if (node.Attributes[parameter] != null)
    transformationInstance.SetParameter(
        parameter, parameter != null ?
        node.Attributes[parameter].Value : null);
}
if (node.Attributes["Priority"] != null)
    transformationInstance.SetPriority(
        node.Attributes["Priority"].Value);
```

The last step to be done is to add the transformation to the *TransformationStore* object. This object serves as an advanced list for storing transformations. The enhancements against a standard list lie in the *AddTransformation()* methods that allow priority based insertion of transformations into the list. Another change is that simple and composite transformations are added in a different way as with composite transformations it is necessary to call the *InstantiateChain()* method to create instances of child transformations:

```
if (IsSubclassOfClass(typeof(CompositeTransformation),
    transformationInstance.GetType())) {
    if (node.Attributes["Priority"] == null)
        store.AddTransformation(((CompositeTransformation)
            transformationInstance));
    else
    store.AddTransformation(((CompositeTransformation)
        transformationInstance),
        transformationInstance.GetPriority());
}
else {
    if (node.Attributes["Priority"] == null)
        store.AddTransformation((Transformation)
            transformationInstance);
    else
        store.AddTransformation((Transformation)
            transformationInstance,
            transformationInstance.GetPriority());
}
```

The *IsSubclassOfClass()* method uses .NET reflection to recursively check for a match in all ancestor classes up to the *Object* class. Reaching the *Object* class signals that we are at the top of inheritance chain as in .NET the *Object* class is the topmost class from which all classes implicitly inherit. This step ends by adding all the transformation objects to the store, by which they become a part of the object model making the XML model unnecessary.

2) Parse and Execute the Metatransformations: After adding the transformations into *TransformationStore*, it is possible to perform metatransformations over this object model. The metatransformations are extracted from all transformations preserving their order as in the store:

```
foreach (Transformation trans in store.Store) {
    foreach (var a in trans.GetMetaTransformations()) {
        string[] parameterNames =
            a.GetParameterNames().ToArray();
        foreach (var parameter in parameterNames) {
            if (trans.GetParameter(parameter) != null)
            a.SetParameter(parameter, parameter != null ?
                trans.GetParameter(parameter) : null );
        }
        if (trans.GetPriority() != null)
            a.SetPriority(trans.GetPriority());
        ...
        // Add the metatransformation to the temporary store
        ...
}
```

The code for adding metatransformation to the temporary metatransformation store is similar to the code regarding common transformations. As a metatransformation can also be a composite transformation, it is again necessary to call the *InstantiateChain()* method. After obtaining a complete metatransformation store, it is possible to proceed with checking the prerequisites and perform the execution of metatransformations:

```
foreach (IMetaTransformation trans in metaStore.Store) {
    trans.CheckPrequisites();
}
foreach (IMetaTransformation trans in metaStore.Store) {
    trans.ExecuteTransformation(model,store);
}
```

After performing this last step, the changes to the transformations contained in the processed metatransformations have been applied to the transformation object model and therefore it is possible to perform the final prerequisite check over the model and proceed with executing the transformations.

3) Check the Requirements of Parsed Transformations: The requirement checking is simple. The only thing that is necessary is to call the *CheckPrerequisites()* method over all the transformations in the *TransformationStore* object. The current implementation uses a simple fault detection mechanism that is based on raising an exception when a problem occurs. One of the signals used is the *TransformationParameterNullException* exception. This signal means that a parameter expected at the XML model level was not provided:

```
public class TransformationParameterNullException : Exception {
    public TransformationParameterNullException( string transID,
        string parameter): base("Transformation" + transID +
            ":Parameter " + parameter + " was not defined."){}
}
```

This exception can be raised afterwards in the *CheckPrerequisites()* transformation method:

```
public override string CheckPrerequisites() {
    if (GetParameter("PageName") == null)
        throw new TransformationParameterNullException(
            this.GetID().ToString(), "PageName");
}
```

With metatransformations it is possible also to check for transformation dependencies using the enhanced model aware method with the necessary parameters: *CheckPrerequisites(**object** model, TransformationStore store).*

4) Execute the Transformations: The precondition for this step is that the *TransformationStore* object contains a list of transformations that is prepared in a way that the metatransformations have been applied and the prerequisites checked. Afterwards, the *ExecuteTransformation()* method is called in a loop for each of the transformations contained in the list:

```
public void ExecuteTransformationChain(
    TransformationStore store) {
    foreach (var transformation in store.Store) {
        transformation.ExecuteTransformation();
    }
}
```

The order in which the transformations are executed is defined by their order in the *TransformationStore* object. After this step, the software artifacts specified in the transformations are created.

VII. Evaluation

To evaluate the approach of feature model driven generation of software artifacts, we developed a study of the family of simple web sites comprising all the possibilities that may arise with features and transformations described in the previous section. Figure 8 shows the corresponding feature diagram.

The page features (ID 2, 3, and 4) embrace two ways of creating the text content: statically, by an HTML document (ID 2 and 3) or dynamically, by a script that generates the HTML document (ID 4).

A model aware transformation is introduced with the dynamic page feature (ID 4), with the utilization of a metatransformation providing the model traversal. Variability is introduced with the data provider (ID 5) providing an XML or Microsoft SQL database backend.

Optional features connected to the root feature (ID 10, 11, and 12) represent crosscutting features that require either metatransformations to change the actual features (ID 10 and 11) or they represent a parameter influencing the generative process (ID 12) to show an implementation of the development or generative environment property.

VIII. Related Work

The pure::variants approach [8], mentioned in Section II, embraces a large set of predefined transformations that are assigned to particular features in the family model. The difference lies in the implementation where pure::variants is relying on XML transformation definitions and the solution proposed here uses C# classes, which is more flexible, allowing for custom programmed transformations.

Edicts [13] is another approach that aims at the mapping of features to source code parts. In addition, Edicts supports different binding times. The concept of binding time [14] should be taken into account when creating feature binding points in the suggested superimposed architectural framework.

XANA [15] strives for bringing closer the development process to end users using feature modeling. It decouples software product line design and implementation, which is to be performed by more technically knowledgeable users or professional developers, from application derivation, which is intended to be manageable by non-technical end users. Application derivation assumes not merely feature selection, but also providing parameters for parameterized features. A similar kind of decoupling can be applied to the approach proposed here. Generic transformations could be provided as a framework. Accompanied by an appropriate development environment extension, these generic transformations could be accessible to end users.

The superimposed variants approach [16] provides a way of mapping features to variabilities in external models, which can be used to activate or deactivate particular parts of the superimposed architectural framework. The transformation based approach proposed here is related to the idea of superimposed variants with respect to the external system of transformations used as the superimposed architecture or model. Differently than in our approach, the superimposed variants approach utilizes external models that are configured [16]. It is also possible to create such transformations that would prepare

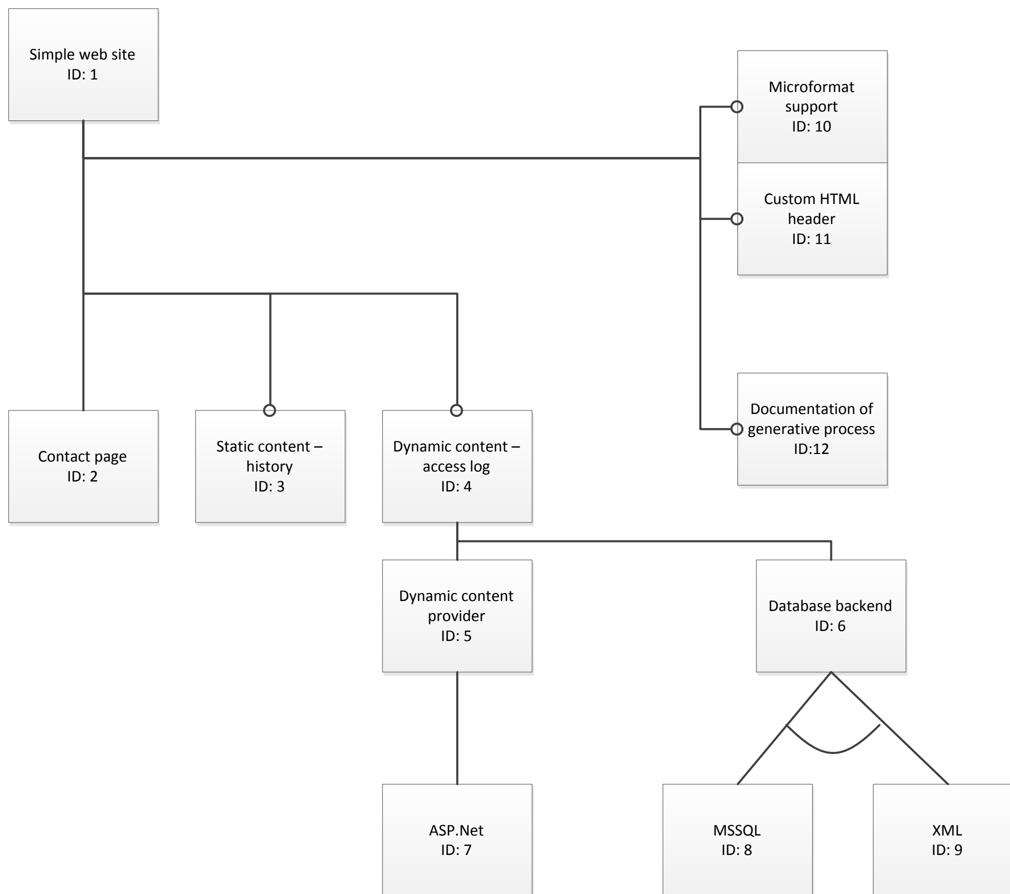

Fig. 8. The feature diagram of the family of simple web sites.

and configure additional models effectively emulating the superimposed variants approach.

One of the actions that is realized by metatransformations in the approach proposed here is the parameter replacement. This is similar to the template text replacement based on generic methods in generative programming for C and C++ [17]. However, the approach proposed here is different in the way that there are no restrictions on what is a transformation template parameter. Another difference is that in a template system, template fields are replaced and then the code is built, but in the approach proposed here, a transformation can represent a complex event chain, and not just a simple text replacement.

Dynamic code structuring [18], [19], [20] is based on explicit representation of possibly overlapping concerns in code for providing different perspectives. In the approach proposed here, dynamic code structuring can be applied to the code that defines transformations. However, dynamic structuring is potentially applicable to feature models themselves. In its essence, featural software decomposition is a decomposition by concerns with features representing the concerns, including the crosscutting ones [12]. Feature models with different organization of features in the feature diagrams can be equivalent [17]. Moreover, a feature can have alternative decompositions into subfeatures, including not being decomposed at all. The different representations of the same feature model may suit different stakeholders or situations and the transformation code attached to it can be presented in different ways accordingly. For large feature model presentation, design pattern detection techniques [21] may be of interest. Feature models can be represented as grammars [22], in which case grammar refactoring could be applied [23] to obtain different views.

IX. Conclusions and Further Work

This paper proposes an approach of feature model driven generation of software artifacts, in which features are viewed as solution space transformations that may be executed during the generative process conducted by the feature model configuration. The approach has been evaluated on a study of the family of simple web sites comprising all the possibilities that may arise with features and transformations.

The main advantages of this approach are is that the system of transformations is basically self-contained and does not require additional modeling techniques except for the enhanced feature model. The code within the transformations is not limited with respect to its effects on the resulting software

system behavior, i.e., anything that can be achieved by manual code writing can be achieved by appropriate transformations. In large part, the flexibility of the proposed approach lies in the concept of metatransformation. Metatransformations are the transformations that represent the impact of crosscutting features by modifying the common transformations before they are executed by changing their input parameters or by modifying their event chains.

A practical adoption of the approach proposed in this paper could be significantly supported by providing directly reusable transformations, transformation templates (i.e., parameterized transformations), or even just transformation schemes or examples to be adapted manually to the application context.

Actual feature models are huge and therefore are more effectively presented by individual concepts [12]. In general, a concept is an understanding of a class or category of elements in a domain [11]. Syntactically, in feature modeling, the root node of a feature diagram represents a concept [17]. Thus, raising a feature to the level of a concept is a matter of choice. Of course, this has to reflect the needs and objectives of the particular case of modeling. Having a feature model decomposed into a set of feature diagrams, rather than a single tree, involves having references between the trees (i.e., concept references [11]). Exploring how this affects feature model driven generation of software artifacts represents a research challenge.

Acknowledgments

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12.

This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA): A feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[2] J. O. Coplien, *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[3] S. Apel, C. Kastner, and C. Lengauer, "FEATUREHOUSE: Language-independent, automated software composition," in *2009 IEEE 31st International Conference on Software Engineering, ICSE 2009*. Vancouver, BC, Canada: IEEE, May 2009. doi: 10.1109/ICSE.2009.5070523 pp. 221–231.

[4] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Proceedings of 4th International Conference on Generative Programming and Component Engineering, GPCE 2005*, ser. LNCS 3676, R. Glück and M. R. Lowry, Eds. Tallinn, Estonia: Springer, Oct. 2005. doi: 10.1007/11561347_28 pp. 422–437.

[5] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multi-level configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, pp. 143–169, Apr./Jun. 2005.

[6] Software Product Line Group, Programming Group, Univeristät Passau, "FeatureHouse: Language-independent, automated software composition," http://www.infosun.fim.uni-passau.de/spl/apel/fh/.

[7] pure-systems GmbH, "pure::variants: Variant management," http://www.pure-systems.com/pure˙variants.49.0.html.

[8] pure systems, "pure::variants user guide," 2015, http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

[9] M. Shaw, "What can we specify? issues in the domains of software specification," in *Proceedings of 3rd International Workshop on Software Specification and Design*. IEEE CS, 1985, pp. 214–215.

[10] P. Sochos, M. Riebisch, and I. Philippow, "The feature-architecture mapping (FArM) method for feature-oriented development of software product lines," in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006, ECBS 2006*. Potsdam, Germany: IEEE, 2006. doi: 10.1109/ECBS.2006.69 pp. 308–318.

[11] V. Vranić, "Reconciling feature modeling: A feature modeling metamodel," in *Proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net.ObjectDays 2004*, ser. LNCS 3263, M. Weske and P. Liggsmeyer, Eds. Erfurt, Germany: Springer, Sep. 2004. doi: 10.1007/978-3-540-30196-7_10 pp. 122–137.

[12] ——, "Multi-paradigm design with feature modeling," *Computer Science and Information Systems Journal (ComSIS)*, vol. 2, no. 1, pp. 79–102, Jun. 2005.

[13] V. Chakravarthy, J. Regehr, and E. Eide, "Edicts: Implementing features with flexible binding times," in *Proceedings of 7th International Conference on Aspect-Oriented Software Development, AOSD '08*. Brussels, Belgium: ACM, 2008, pp. 108–119.

[14] V. Vranić and M. Šípka, "Binding time based concept instantiation in feature modeling," in *Proceedings of 9th International Conference on Software Reuse, ICSR 2006*, ser. LNCS 4039, M. Morisio, Ed. Turin, Italy: Springer, Jun. 2006. doi: 10.1007/11763864_34 pp. 407–410.

[15] V. Tzeremes and H. Gomaa, "A software product line approach for end user development of smart spaces," in *Proceedings of 5th International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2015*. IEEE, 2015. doi: 10.1109/PLEASE.2015.14 pp. 23–26.

[16] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Proceedings of 4th International Conference on Generative Programming and Component Engineering, GPCE 2005*, ser. LNCS 3676, 2005. doi: 10.1007/11561347_28 pp. 422–437.

[17] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] M. Nosáľ and J. Porubän, "Supporting multiple configuration sources using abstraction," *Central European Journal of Computer Science*, vol. 2, no. 3, pp. 283–299, 2012. doi: 10.2478/s13537-012-0015-7

[19] M. Nosáľ, J. Porubän, and M. Nosáľ, "Concern-oriented source code projections," in *Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013*. Kraków, Poland: IEEE, 2013, pp. 1541–1544.

[20] J. Porubän and M. Nosáľ, "Leveraging program comprehension with concern-oriented source code projections," in *Proceedings of Slate'14, 3rd Symposium on Languages, Applications and Technologies*, Bragança, Portugal, 2014. doi: 10.4230/OASIcs.SLATE.2014.35 pp. 35–50.

[21] I. Polášek, P. Líška, J. Kelemen, and J. Lang, "On extended similarity scoring and bit-vector algorithms for design smell detection," in *Proceedings of 2012 IEEE 16th International Conference on Intelligent Engineering Systems, INES 2012*. Lisbon, Portugal: IEEE, 2012. doi: 10.1109/INES.2012.6249814 pp. 115–120.

[22] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005. doi: 10.1002/spip.213

[23] J. Kollár, I. Halupka, S. Chodarev, and E. Pietriková, "pLERO: Language for grammar refactoring patterns," in *Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013*. Kraków, Poland: IEEE, 2013, pp. 1491–1498.