Computer Science and Information Systems 13(3):759–778 DOI: 10.2298/CSIS160128027V

Features as Transformations: A Generative Approach to Software Development

Valentino Vranić and Roman Táborský

Institute of Informatics, Information Systems and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, 84216 Bratislava 4, Slovakia
vranic@stuba.sk, crudecrude@gmail.com

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 is used to configure other models and, eventually, code. 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. Thus, 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 transformations that may be executed during the generative process conducted by the feature model configuration. The generative process is distributed in respective transformations enabling the developers to have a better control over it. This approach can be applied to modularize changes in product customization and to establish generative software product lines by gradual refactoring of existing products.

Keywords: feature modeling, transformation, metatransformation, generative process, reuse, change, customization, software product lines, variability.

1. Introduction

Feature modeling is a widely known approach to capturing commonality and variability proposed in 1990's [18]. Although industrial applications of its academic form (not counting in some cases of using the pure::variants notation and tool [5,6,4]) are unknown [17], it is widely recognized that commonality and variability lie at heart of organizing software development for reuse in software product lines. Based on commonality and variability, appropriate implementation mechanisms can be selected [9].

Feature modeling is used to configure other models and, eventually, code. 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, a feature inclusion affects several places in models or code that belong to other features, requiring their nontrivial adaptation or—to use a more appropriate word—their *transformation*. This is known as feature interaction. In his keynote

at Modularity 2016 [1], Sven Apel admitted that only after years of fighting it, he found feature interaction is inevitable and actually necessary. By perceiving features as transformations, resolving feature interaction becomes their intrinsic part rather than something that has to be tackled externally.

Modularizing other models and code by features has been recognized as a viable approach to implementing features in some of the so-called language based approaches, such as GenVoca or AHEAD, that fall into the category of feature-oriented programming [2]. However, such approaches require language extensions, which are hard to keep compatible with the ever changing host language without a dedicated support.

Another way to see feature interaction is crosscutting in the sense of aspect-oriented programming, but aspect-oriented programming extensions that are stable and up-to-date with the host language are not available for most programming languages. On the other hand, crosscutting features have been reported to be perceived as problematic by developers [4]. Yet, a significant number of features exhibit a crosscutting nature [4] and such features are often highly important [16].

The approach to feature model driven generation of software artifacts proposed in this paper is intended to overcome the problems mentioned above. It does so by providing a self-contained transformation framework realizable in a general purpose object-oriented programming language capable of making model aware interventions and handling crosscutting features by modifying the transformations that implement other features. The generative process is distributed in respective transformations enabling the developers to have practically unlimited control over it given by the power of the general purpose programming language being used. At the same time, the developers have to implement and maintain only that much of the generative process that is really necessary.

The rest of the paper[1] is organized as follows. Section 2 offers a view on features as means for software modularization. Section 3 explains how features can be perceived as transformations, which is the essence of the approach proposed in this paper. Section 4 presents the implementation of the approach in a form of a framework. Section 5 demonstrates the application of the approach. Section 6 discusses how the approach can be applied in software product customization viewing changes as transformations and how this can be extended to software product lines. Section 7 discusses related work. Section 8 concludes the paper.

2. Modularizing Software by Features

While there is no perfect software modularization and, ultimately, any kind of modularization is condemned to tyrannize some developers or other stakeholders [29], features are a particularly interesting form of software modularization from the perspective of the application domain. Simply stated, a feature is an important property of a given concept and any concept itself can be perceived as a feature of another concept [34]. Roughly speaking, features imply from requirements and domain analysis [10], but feature modeling actually does not deal with features themselves, but with their relationships. These may be captured graphically by feature diagrams. In feature diagrams, features are usually

[1] This paper is an extended version of the paper selected among the papers presented at WAPL 2015 [30].

organized hierarchically into trees, being easier for humans to follow trees than general graphs, features are usually organized hierarchically into trees while additional relationships are expressed textually [34,35].

Figure 1 shows an example feature diagram in Czarnecki–Eisenecker notation. Edges and arc decorations denote the kind of feature variability: a filled circle ended edges stand for mandatory features, an empty circle ended edges stand for optional features, and an arc groups alternative features. A feature diagram defines a whole family of products or product parts. Each member of a family is defined by a configuration of the features. Figure 2 shows several valid configurations of the simple web site, including the minimal one (with only one feature besides the root one). Depending on its variability constraints, a feature may or may not be included in a configuration. A feature may not be included in a configuration if its parent feature is not included.

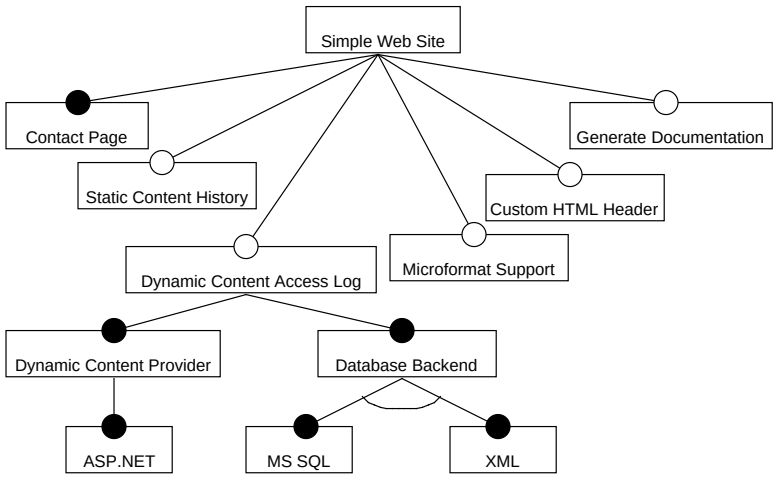

Fig. 1. The feature diagram of the family of simple web sites.

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 [11,15]. The FeatureHouse project [27] implements an approach that uses these models and allows language independent source code generation.

The opposite way is to generate or add pre-built software artifacts according to the features contained in the configuration. The pure::variants software tool [25] 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.

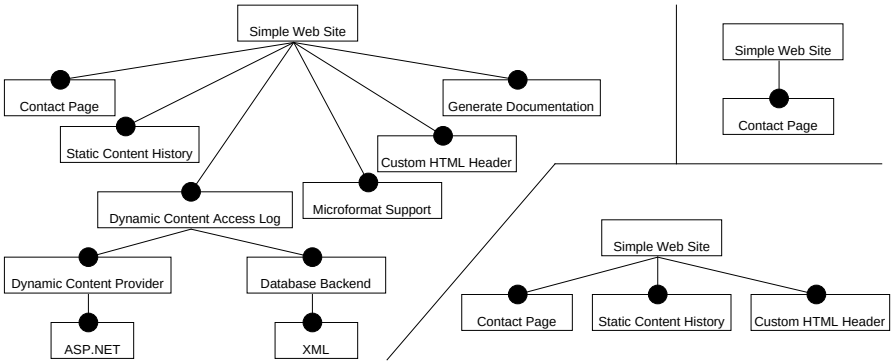

Fig. 2. Several simple web site configurations.

Features can represent anything. At some point, one may be tempted to use them to extensively model structural concerns. The logical next step would then be to extend feature modeling with multiplicities (cardinalities) and this has actually happened [13]. However, the original perception of features is different: a feature is like a proposition and to include a feature several times would simply mean to state the same thing several times [10]. This keeps feature models focused on the configurability aspect and is in accordance with viewing features as transformations. While transformations, in general, may be repeatable and their repetition may even make sense, that would unnecessarily complicate the whole model. The same effect may be achieved by transformation parameters one of which may be the number of times to repeat the transformation.

3. Features as Transformations

Generating software artifacts independently for each included feature may be sufficient only for simple situations. In reality, features may interact and this interaction may result in the necessity to generate different software artifacts for the affected features or to change them if they are already generated. But changing already built things may require deep restructuring. To compare to more tangible artifacts, once a sculpture is cast, it is very hard to modify it without making this obvious. A smoothly modified instance of the sculpture can be obtained only by modifying the cast itself. Similarly, instead of modifying generated software artifacts, in order to get a smooth result, it is more appropriate to modify the procedure that generated them. These procedures actually transform the premanufactured software material into its final form. If each feature is viewed as a *transformation*, this can be taken one step further: the transformation associated with the affecting feature could modify the transformations associated with the affected features instead of harshly modifying already generated software artifacts.

Although transformations will mainly have to result in generated functional code, a transformation can represent virtually any kind of action, including those that reflect extra-

functional requirements.[2] Thus, it is legitimate to have a transformation that will provide logging or documentation.

This section explains the overall process of our approach to feature model driven generation of software artifacts based on features as transformations (Section 3.1) and its different aspects (Sections 3.2–3.5).

3.1. The Overall Process

The overall process of the approach to feature model driven generation of software artifacts based on features as transformations is depicted in Figure 3. The whole process actually consists of two processes one of which is the transformation design, and the other one is their application. This is analogous to distinguishing domain engineering from application engineering [10].

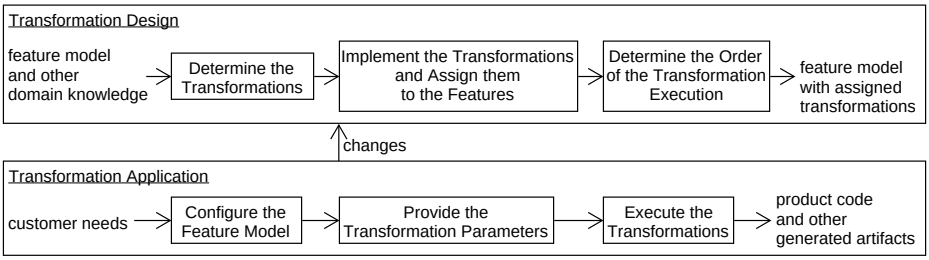

Fig. 3. The overall process.

The input to the transformation design process is a feature model and other domain knowledge. This process embraces determining the transformations that are necessary, implementing these transformations and assigning them to the corresponding features, and determining the order in which the transformations are to be executed. Although not indicated in Figure 3, this process is iterative and incremental.

The process of the transformation application starts with the customer needs, which determine what features are to be included in the corresponding feature model configuration. In turn, the selection of the features determines the set of the transformations to be executed. The corresponding parameters are provided to the transformations and, finally, the transformations are executed to obtain the product code and other generated artifacts such as documentation.

Again, this process is also iterative and incremental requiring to experiment with the selection of the features that suits best the customer needs. This embraces experimenting with transformation parameters, too. Moreover, the need for changes may also propagate into the transformation design process and even affect the feature model itself. Experimenting with transformation application is of particular importance in assuring the order of their application is correct.

[2] We use term *extra-functional*—as proposed by Mary Shaw [26]—to refer to requirements and features that go beyond software system functionality instead of more widely used, but potentially confusing term non-functional.

3.2. Metatransformations

Some features may 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. Other features rooted in extra-functional requirements often have such effect, too. Including such a feature with the corresponding transformation into the configuration leads to the necessity of accessing or modifying other transformation parameters or actions. We denote the transformations capable of this as *metatransformations*. They correspond to the concept of higher-order transformation [31] and in particular to two kinds of higher-order transformations: transformation modification, with respect to modifying transformation actions or parameters, and transformation analysis, with respect to accessing transformation actions or parameters. The notion is also related to the concept of model metatransformations [33].

3.3. Associating Transformations with Features

One way of associating features with transformations is to include the transformations directly in the feature model. 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 may be the same (e.g., create a file).

Another approach is to store transformation definitions outside the feature model as demonstrated by this C# class:

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

A feature model then includes only the association itself:

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

Although a feature appears to be associated with the transformation class as such, it is rather its instance it is associated with. This way, in addition to the *actions* to be performed within the transformation common to all its instances, each such *transformation instance* can have its own, independent state to operate upon. This state will usually depend on *transformation parameters*, such as file or folder name in the creating a file or folder transformation.

What transformations are to be executed is given by the selection of features in a feature model configuration. It is obvious that the order of execution is significant. The order may be defined externally, i.e., globally for the whole feature model, or internally, i.e., locally for each feature. An internally defined order, where each feature defines what other features have a priority over it, overcomes the necessity of a global rearrangement of the feature order upon a change in feature model, which is unavoidable with the externally defined order.

3.4. Transformation Granularity and Modularization

Besides the features used merely to group other features under a common denominator, which might have no corresponding transformations, such as *Dynamic Access Log* in our family of simple web sites (recall Figure 1), any other feature would normally be associated with a transformation.

This is in accordance with the separation of concerns principle, but may lead to concern fragmentation which may cause difficulties in concern comprehension. To avoid this, a general concern may be captured in one transformation modified by the metatransformations assigned to variable features depending on their selection in the feature model configuration process. An extreme of this would be having only one general transformation presumably associated with the root feature.

On the other side, a transformation associated with one feature may actually comprise several transformations that, in turn, may be individually associable with other features. In general, such a *composite transformation* may invoke the transformations it comprises within the its course of actions unconditionally, conditionally, or repeatedly. These transformations can be composite themselves.

3.5. Transformation Reusability

The process of the transformation design requires 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 reusability levels of transformations:

- Specialized transformations that can be used only for the specific features in a 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

As with common object-oriented programming, inheritance can be used to realize some aspects of transformation reuse. In this, actions play the role of methods and a devised transformation can preserve the inherited actions, add new ones, remove some of them, or override them.

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 a 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.

4. The Framework

We implemented the proposed approach to feature model driven generation of software artifacts based on features as transformations as a framework in .NET using C#. The approach itself is by no means limited to .NET. Also, can be implemented in other object-oriented programming languages.

The overall architecture is depicted in Figure 4. The details are explained in Sections 4.1–4.5.

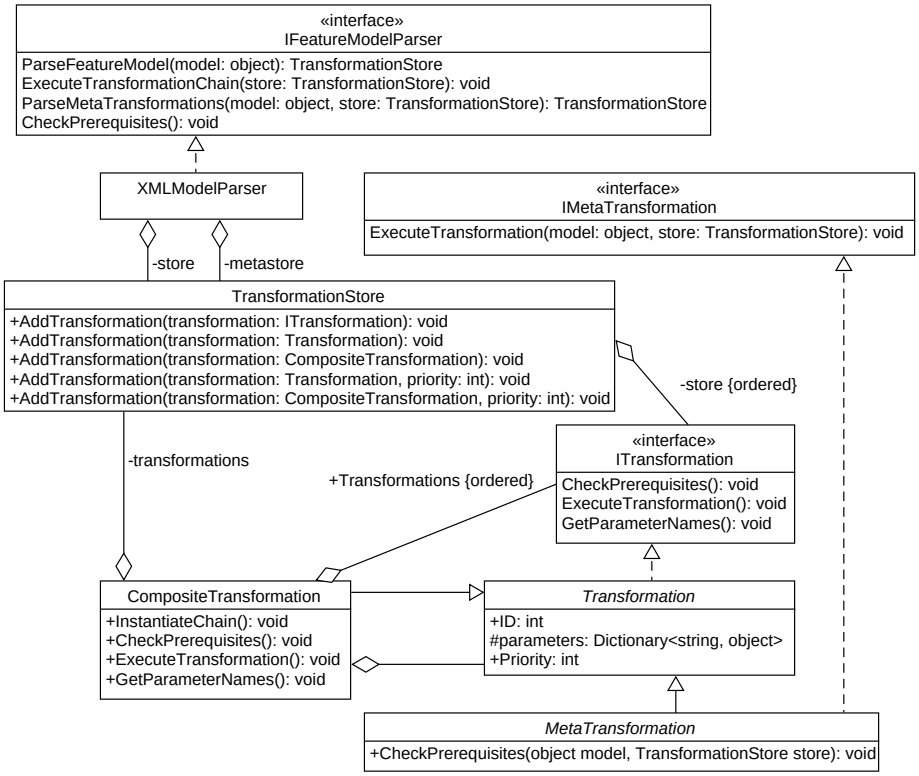

Fig. 4. The overall architecture.

4.1. Transformations

All transformations in our implementation are objects of the *Transformation* class that implements the *ITransformation* interface. The *CheckPrerequisites()* method performs some basic parameter checking before the transformation is processed. In general, it is possible to provide a model aware method at the level of a metatransformation that can check the transformation dependencies, too.

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 to provide the parameter names to be retrieved from the feature node attributes.

The *GetMetaTransformations()* method provides a way 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.

The parameters are stored in the dictionary format in the *parameters* attribute.

4.2. Composite Transformations

A composite transformation contains an object of the *TransformationStore* class, which is basically an ordered list of transformations. For processing these transformations, the *CheckPrerequisites()* and *ExecuteTransformation()* methods have to be overridden. The contained transformations are instantiated by overriding the *InstantiateChain()* method. This method is automatically executed by the *TransformationStore* object when adding the transformation to it during parsing. The contained transformations may themselves be composite. As it is obvious from the implementation, such transformations are processed recursively in a depth-first order.

4.3. 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.

For this, the *ExecuteTransformation method()* method takes as an argument all transformations within the *TransformationStore* object. To allow for the modification of composite transformations, this method is implemented in a recursive way.

4.4. Feature Model Configuration

In our implementation, feature model configurations are represented by XML files. The structure of these files 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 it is being 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 example:

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

The XML feature model configuration is afterwards transformed into an object model contained within the *TransformationStore* object.

4.5. Parsing and Executing Transformations

The generative process consists of these steps:

1. Parse the feature model configuration from the XML file
2. Parse and execute the metatransformations contained within the transformation from the XML file
3. Check the parameters of the parsed transformations
4. Execute the transformations

The step of checking the parameters before parsing the metatransformation is omitted as it is contained within the metatransformation parsing step. The actual generation of code and other artifacts is performed by the transformations making the whole generative process distributed.

An abstract prescription of these steps is provided in a form of an interface denoted as *IFeatureModelParser*. As has been mentioned in Section 4.4, in our implementation, feature model configurations are represented by XML files. Therefore, we implemented the corresponding XML parser and transformation executor as the *XMLModelParser* class. We will refer to it simply as ”parser” in the following text.

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 the transformation instance is created using reflection:

```
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));
```

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

The transformations are stored in a list implemented by the *TransformationStore* class. Its *AddTransformation()* method allows for priority based insertion of transformations. The transformation type is preserved in order to enable distinguishing composite transformations during their execution.

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 hierarchy 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.

After adding the transformations into *TransformationStore*, it is possible to perform metatransformations over this object model. The metatransformations are separated from all transformations preserving their order as in the transformation store.

After obtaining a complete metatransformation store, it is possible to proceed with checking the prerequisites and perform the execution of metatransformations. Afterwards, 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.

To check their parameters, the *CheckPrerequisites()* method is executed for each transformation in the *TransformationStore* object. The current implementation uses a simple fault detection mechanism that is based on raising an exception when a problem occurs. With metatransformations, it is possible also to check for transformation dependencies using the enhanced model aware *CheckPrerequisites()* method with the necessary parameters.

To execute the transformations, the *TransformationStore* object has to contain 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. The order in which the transformations are executed is defined by their order in the *TransformationStore* object. Their execution finally creates the corresponding software artifacts.

5. Applying the Approach

To demonstrate its applicability, we used the framework for feature model driven generation of software artifacts based on features as transformations described in the previous section to develop a family of simple web sites as present in Section 5.1. We have also applied the ideas behind the approach in practice described in Section 5.2. The discussion is provided in Section 5.3.

5.1. Using the Framework

The feature diagram of the family of simple web sites has been introduced in Figure 1. Variability includes having static content history, dynamic content access log, microformat support, custom HTML header, and generated documentation.

The actual content is provided by transformation parameters. Consider the *CreateStaticHTMLPage* transformation as an example. This transformation embraces a prototype HTML file in its *htmlContent* attribute:

```
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>";
```

This transformation has four parameters used to adjust the prototype HTML file:

- *PageName*: the file name under which the HTML file will be created
- *htmlTemplate*: the template that is set up can be customized with this parameter
- *htmlTitle*: the text that replaces the *Title placeholder* text
- *htmlContent*: the text that replaces the *Content placeholder* text (different from the *htmlContent* attribute explained above)

If the corresponding feature to which this transformation is associated is included in the feature model with the following parameter configuration (with *htmlTemplate* left default):

```
<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" />
```

the following HTML file will be created:

```
<!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>
```

The page features—*Contact Page*, *Static Content History*, and *Dynamic Content Access Log*—embrace two ways of creating the text content: statically, by an HTML document, as with the *Contact Page* and *Static Content History* features, or dynamically, by a script that generates the HTML document, as with the *Dynamic Content Access Log* feature. The *Static Content History* feature is implemented as a composite transformation.

A model aware transformation is introduced with the dynamic page feature, i.e., the *Dynamic Content Access Log* feature, with the utilization of a metatransformation providing the model traversal. Variability is introduced with the data provider, i.e., the *Dynamic Content Provider* feature, providing an XML or Microsoft SQL database backend.

The optional features connected to the root feature—*Microformat Support*, *Custom HTML Header*, and *Generate Documentation*—represent crosscutting features: they affect other features not directly connected with them. The transformations for such features either have to comprise the metatransformations to change the respective transformations, as it is with the *Microformat Support* and *Custom HTML Header* features, or they can appear as parameters to influence these transformations, such as the *Generate Documentation* feature.

5.2. Applying the Approach in Practice

We applied the ideas of the approach to feature model driven generation of software artifacts based on features as transformations in practice in three industrial projects. The application was partial and emergent. It concerned security management in one project and partially in two other projects in ASP.NET emerging out of the needs for reuse. Although no explicit feature model was created during the application itself, a featural decomposition was applied. Also, no dedicated framework was used, so the transformations, implemented in T-SQL and XSLT, were executed manually.

Figure 5 shows a reconstructed feature model. It abstracts from platform specific features and corresponding transformations such as modifications of the Visual Studio solution and project files.

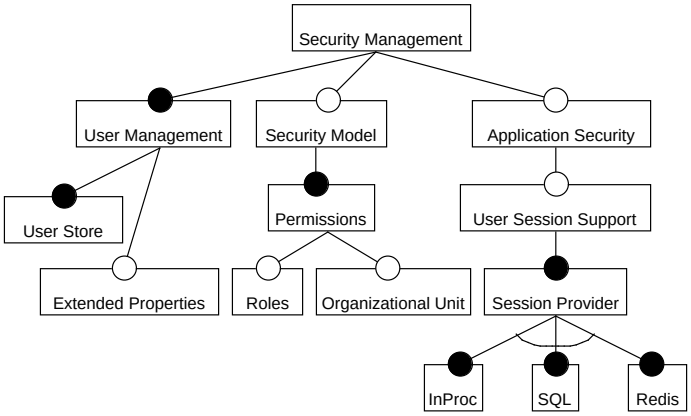

Fig. 5. The feature diagram of the security management subsystem.

Security Management is the top level feature that represents the whole subsystem. It implements the transformation that collects the T-SQL scripts, deploys them to database, and then generates the ORM layer.

User Management represents the management of the user and password store. No transformation is associated directly with this feature, but there are transformations associated with its subfeatures. *User Store* represents the data store for user records. This feature could be extended by allowing for more types of data stores. Currently, only T-SQL is used. The transformation that generates the *CREATE TABLE* script for the user store is associated with it. *Extended Properties* is an optional feature that stands for the addition of custom properties of user records. A composite transformation that consists of three transformations that create scripts for regulating custom properties is associated with it.

Security Model stands for what its name says. No transformation is associated directly with this feature, but a composite transformation is associated with each of its subfeatures. These composite transformations follow a common pattern consisting of three transformations to generate the scripts that create the corresponding database tables. Thus, the *Permissions* feature regulates user permissions, *Roles* covers the addition of roles to manage permissions and *Organizational Unit* manages the relation of users to organizational units. In addition, the metatransformation altering the transformation associated with *User Store* to add the role column is associated with the *Roles* feature.

Application Security controls in which way the user is authenticated. Composite transformations that copy files from template files are associated with it. *User Session Support* is only a container feature that allows to choose a session provider based on its optional subfeatures. With *InProc* no action is performed (the default provider), while *SQL* and *Redis* activate the metatransformations that alter the files copied in the *Application Security* feature transformation.

5.3. Discussion

The two example applications presented in the previous two sections demonstrated the approach applicability. With respect to reusability, since our web site family example comprised only one feature model, by the means of implementation, we actually proved achieving only the level of specialized transformations. These can be used only for the specific features, but in any configuration of the feature model (recall Section 3.5). However, several transformations such as those associated with *Database Backend* or *Generate Documentation*, are potentially domain dependent generic transformations or even domain independent generic transformations.

We go beyond this with the security management example. The whole set of transformations associated with it is actually at the domain dependent generic level proven by their use in three projects.

The validity of these claims is on one side threatened by the small extent of the web site family example, while on the other hand, the security management example comprises no supporting framework with transformations being executed manually. However, the two applications are complementary with each one targeting what is weak in the other one. Also, it is important to note that both examples embrace the application of all three kinds of transformations: general transformations, composite transformations, and metatransformations.

6. Changes as Transformations

So far, we assumed an exclusive application of the approach to feature model driven generation of software artifacts based on features as transformations. However, it is also possible to apply this approach non-exclusively and non-primarily using an existing code base, which, in general, would not be modularized by features. In such case, the features to be added could be viewed as changes. It is not necessary to create a complete feature model for this: the technique of partial feature modeling can be applied [38,20]. In the first line, only the new features would be singled out. Consider again our simple web site. Assume it was developed manually, perhaps with all the features in the feature model presented in Figure 1 and that this feature model was never actually created. It is not known to which features the changing features are to be attached and this is indicated by a dashed line.

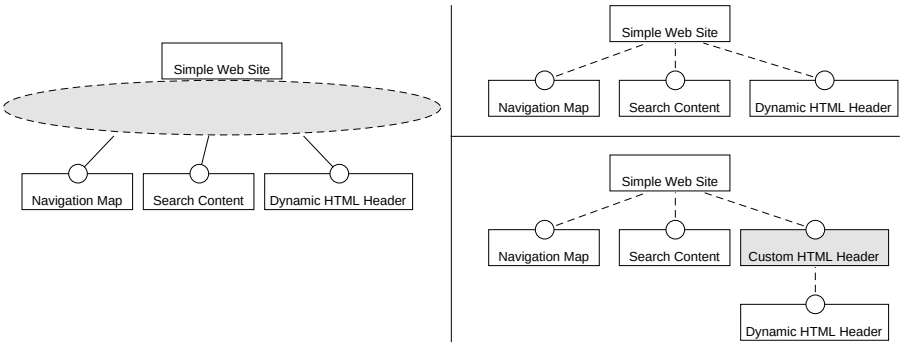

Fig. 6. A partial feature model: extending the simple web site with new features.

Having changes modularized is very useful in product customization, where a general application is being adapted to the client's needs by a series of changes [37]. These changes have to be reapplied to each new version as schematically indicated in Figure 7. It has been demonstrated how such changes could be modularized by implementing them as aspects [37]. Similar effects may be achieved by modularizing changes as transformations. This approach is potentially applicable where aspect-oriented programming is not available or not appropriate.

Intermediate features can be factored out and reimplemented as transformations, too. For example, *Dynamic HTML Header*, one of our change features (recall Figure 1), can be found out to be a *Custom HTML Header* variant uncovering this feature. In effect, this is gradual refactoring of a monolithic software product into a generative software product line. This process can be conducted without considering any changes to the initial product or even a set of products. This way of thinking would lead to identifying the most obvious common features. However, if the process is driven by the need to adjust the product to accommodate the variability required by clients, these features become variable. This can be considered as an economically feasible way of establishing a software product line not requiring a big upfront investment as would be necessary if the software product line would be developed from scratch [7].

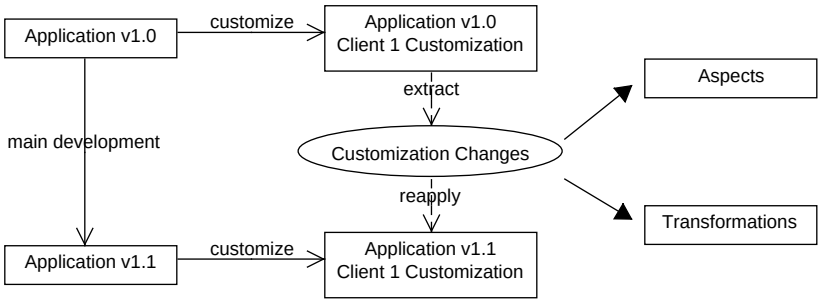

Fig. 7. Customization [36] (adapted).

7. Related Work

In this section we attempt to put our approach to feature model driven generation of software artifacts based on features as transformations into the context of related work. We have identified several approaches that address similar issues or that appear to be complementary to our approach.

The pure::variants approach [28], mentioned in Section 2, embraces a large set of predefined transformations that are assigned to individual features in the family model. The difference lies in the implementation: pure::variants relies on XML transformation definitions, and the solution proposed here uses C# classes. While XML is only a markup language, C# is a full-fledged programming language allowing for the transformations to gain the full control over their target. The approach itself is not bound to C#: other programming languages can be used instead.

Edicts [8] is another approach that aims at the mapping of features to source code parts. In addition, Edicts supports variable binding times allowing for optimal solutions for different contexts. In general, earlier binding times are more time efficient, while later binding times offer a greater flexibility up to runtime feature selection [39]. In our approach, this concern could be addressed by metatransformations.

XANA [32] 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 could be applied in our approach: generic transformations could be provided as a framework, while being accompanied by an appropriate development environment extension to make them accessible to end users for selection.

The superimposed variants approach [12] 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. Our transformation based approach 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 being configured [12]. An

analogy in our approach would be to create such transformations that would prepare and configure additional models.

One of the actions that is realized by metatransformations in our approach is parameter replacement. This is similar to the template text replacement based on generic methods in generative programming for C and C++ [10]. Moreover, in our approach, a transformation can represent a complex set of actions, and not just a simple text replacement.

Dynamic code structuring [22,21,24] is based on explicit representation of possibly overlapping concerns in code for providing different perspectives, which can help in preserving intent comprehensibility [40]. In our approach, 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 concerns, including the crosscutting ones [35]. Feature models with different organization of features in feature diagrams can be equivalent [10]. Moreover, a feature can have alternative decompositions into subfeatures, including not being decomposed at all. 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.

Our approach employs the basic Czarnecki–Eisenecker notation, but it is not limited to it. Feature models can be represented as grammars [14], in which case grammar refactoring could be applied [19] to obtain different views. For dealing with large feature models, design pattern detection techniques [23] may be of interest.

8. Conclusions and Challenges

This paper proposes an approach to 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 was implemented as a framework in .NET. Two examples of the approach application were presented one of which constitutes a practical application.

The main advantages of this approach are that the transformation framework 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 transfer the impact of crosscutting features to the implementation level by modifying the transformations associated with the features being crosscut before they are executed by changing their input parameters or by modifying their actions.

There are several research challenges with respect to having features implemented as transformations. A practical adoption of the approach 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. It should be explored further how the approach could be utilized in establishing software product lines out of existing products. With respect to this, implementing use cases as transformations could be an interesting research direction. In particular, this is related to extension use cases, which exhibit a crosscutting nature.

Actual feature models are huge and therefore are more effectively presented by individual concepts [35]. In general, a concept is an understanding of a class or category of elements in a domain [34]. Syntactically, in feature modeling, the root node of a feature diagram represents a concept [10]. Thus, raising a feature to the level of a concept is a matter of choice. Having a feature model decomposed into a set of feature diagrams involves maintaining the references between these diagrams (i.e., concept references [34]). It is necessary to explore how this affects feature model driven generation of software artifacts.

Acknowledgments. The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/0808/17. 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. Apel, S.: From crosscutting concerns to feature interactions: A tale of misunderstandings and enlightenments (keynote). In: MODULARITY Companion 2016, Companion Proceedings of the 15th International Conference on Modularity, Modularity 2016. ACM, Málaga, Spain (2016)
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation, chap. Advanced, Language-Based Variability Mechanisms, pp. 129–174. Springer (2013)
3. Apel, S., Kastner, C., Lengauer, C.: FEATUREHOUSE: Language-independent, automated software composition. In: 2009 IEEE 31st International Conference on Software Engineering, ICSE 2009. IEEE, Vancouver, BC, Canada (2009)
4. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature? a qualitative study of features in industrial software product lines. In: Proceedings of 19th International Software Product Line Conference, SPLC '15. ACM, Nashville, TN USA (2015)
5. Berger, T., Nair, D., Rublack, R., Atlee, J.M., Czarnecki, K., Wąsowski, A.: Three cases of feature-based variability modeling in industry. In: Proceedings of ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MODELS 2014. vol. LNCS 8767. Springer, Valencia, Spain (2014)
6. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wasowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of 7th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2013. ACM (2013)
7. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley (2000)
8. Chakravarthy, V., Regehr, J., Eide, E.: Edicts: Implementing features with flexible binding times. In: Proceedings of 7th International Conference on Aspect-Oriented Software Development, AOSD '08. ACM, Brussels, Belgium (2008)
9. Coplien, J.O.: Multi-Paradigm Design for C++. Addison-Wesley (1999)
10. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
11. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M.R. (eds.) Proceedings of 4th International Conference on Generative Programming and Component Engineering, GPCE 2005. pp. 422–437. LNCS 3676, Springer, Tallinn, Estonia (2005)

12. Czarnecki, K., Antkiewicz, M.: 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. LNCS 3676 (2005)
13. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In: 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002. LNCS 2487, Pittsburgh, PA, USA (2002)
14. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)
15. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process: Improvement and Practice 10, 143–169 (2005)
16. Ferber, S., Haag, J., Savolainen, J.: Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In: Proceedings of 2nd International Software Product Line Conference, SPLC 2. Springer, San Diego, CA, USA (2015)
17. Hubaux, A., Classen, A., Mendonça, M., Heymans, P.: A preliminary review on the application of feature diagrams in practice. In: Proceedings of 4th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2010. ICB Research Report 37 (2010)
18. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA): A feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (1990)
19. Kollár, J., Halupka, I., Chodarev, S., Pietriková, E.: pLERO: Language for grammar refactoring patterns. In: Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013. pp. 1491–1498. IEEE, Kraków, Poland (2013)
20. Menkyna, R., Vranić, V.: Aspect-oriented change realization based on multi-paradigm design with feature modeling. In: Proc. of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009. LNCS 7054, Springer, Kraków, Poland (2009)
21. Nosáľ, M., Porubän, J., Nosáľ, M.: Concern-oriented source code projections. In: Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013. pp. 1541–1544. IEEE, Kraków, Poland (2013)
22. Nosáľ, M., Porubän, J.: Supporting multiple configuration sources using abstraction. Central European Journal of Computer Science 2(3), 283–299 (2012)
23. Polášek, I., Líška, P., Kelemen, J., Lang, J.: 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. pp. 115–120. IEEE, Lisbon, Portugal (2012)
24. Porubän, J., Nosáľ, M.: Leveraging program comprehension with concern-oriented source code projections. In: Proceedings of Slate'14, 3rd Symposium on Languages, Applications and Technologies. pp. 35–50. Bragança, Portugal (2014)
25. pure-systems GmbH: pure::variants: Variant management, `http://www.pure-systems.com/pure_variants.49.0.html`
26. Shaw, M.: What can we specify? issues in the domains of software specification. In: Proceedings of 3rd International Workshop on Software Specification and Design. pp. 214–215. IEEE CS (1985)
27. Software Product Line Group, Programming Group, Univeristät Passau: FeatureHouse: Language-independent, automated software composition, http://www.infosun.fim.uni-passau.de/spl/apel/fh/
28. pure systems: pure::variants user guide (2015), http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf
29. Tarr, P., Ossher, H., Harrison, W., Staneley M. Sutton, J.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of 21st International Conference on Software Engineering, ICSE'99. ACM (1999)

30. Táborský, R., Vranić, V.: Feature model driven generation of software artifacts. In: Proceedings of 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015. IEEE, Łódź, Poland (2015)
31. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) Model Driven Architecture – Foundations and Applications, Proceedings of 5th European Conference, ECMDA-FA 2009. LNCS 5562, Springer, Enschede, The Netherlands (2009)
32. Tzeremes, V., Gomaa, H.: 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. pp. 23–26. IEEE (2015)
33. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 214–234 (2007)
34. Vranić, V.: Reconciling feature modeling: A feature modeling metamodel. In: Weske, M., Liggsmeyer, P. (eds.) Proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net.ObjectDays 2004. pp. 122–137. LNCS 3263, Springer, Erfurt, Germany (2004)
35. Vranić, V.: Multi-paradigm design with feature modeling. Computer Science and Information Systems Journal (ComSIS) 2(1), 79–102 (2005)
36. Vranić, V.: Aspect-Oriented Change Realization. Habilitation thesis, Slovak University of Technology in Bratislava, Slovakia (2010)
37. Vranić, V., Bebjak, M., Menkyna, R., Dolog, P.: Developing applications with aspect-oriented change realization. In: Proceedings of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2008, Revised Selected Papers. LNCS 4980, Springer, Brno, Czech Republic (2011)
38. Vranić, V., Menkyna, R., Bebjak, M., Dolog, P.: Aspect-oriented change realizations and their interaction. e-Informatica Software Engineering Journal 3(1), 43–58 (2009)
39. Vranić, V., Šípka, M.: Binding time based concept instantiation in feature modeling. In: Morisio, M. (ed.) Proceedings of 9th International Conference on Software Reuse, ICSR 2006. pp. 407–410. LNCS 4039, Springer, Turin, Italy (2006)
40. Vranić, V., Porubän, J., Bystrický, M., Frt'ala, T., Polášek, I., Nosáľ, M., Lang, J.: Challenges in preserving intent comprehensibility in software. Acta Polytechnica Hungarica 12(7), 57–75 (2009)

Valentino Vranić is an associate professor of software engineering at the Slovak University of Technology in Bratislava. He explores different aspects of software development. In particular, he is interested in preserving intent comprehensibility in code and models using advanced modularization, as well as in effective agile and lean organization of people in software development and its wider social connotations.

Roman Táborský received an MSc. in software engineering from the Slovak University of Technology in Bratislava. He works as a software developer aiming at exploiting the ideas of generative approaches in practice.

Received: January 28, 2016; Accepted: September 29, 2016.