Toward Automated Refactoring of Feature Models using Graph Transformations *

Sergio Segura, David Benavides, Antonio Ruiz-Cortés and Pablo Trinidad

Department of Computer Languages and Systems

University of Seville

Av. de la Reina Mercedes S/N, 41012 Seville, Spain

e-mail: {segura, benavides, aruiz, trinidad}@tdg.lsi.us.es

Abstract

Software Product Line (SPL) development is an approach to develop families of software systems in a systematic way. A Feature Model (FM) represent all the products in a SPL in terms of features. Applying refactoring to an SPL can have the highly negative effect of reducing the number of differents products in the SPL. Hence, it is accepted that not only programs must be refactored but also its associated FMs. In this paper we present a first proposal for automated support for FM refactoring based on graph transformations. We also explain how we plan to integrate our proposal in the FAMA plug-in and we clarify our contribution with an example.

Keywords: Feature Models, Refactoring, Feature Oriented Programming, Graph Transformations.

1 Introduction and Motivation

Software Product Line (SPL) development [12] is an approach to develop families of software systems in a systematic way. Roughly speaking, an SPL can be defined as a set of software products that share a common set of features. A Feature Model (FM) represents all possible products in an SPL in terms of features. A feature is an increment in product functionality. FMs can be used in different stages of SPL development such as requirements engineering [17], architecture definition [22] or code generation [5].

The term *refactoring* refers to the changes made to a program in order to enhance its structure in some way while preserving its behaviour [15, 21]. However, the traditional definitions of refactoring do not cover the intrinsic characteristics of SPLs in which programs are frequently derived from a FM representing all the products in the SPL. In [1] Alves *et al.* study refactoring in the context of SPL and conclude that not only programs should be refactored but also its associated FMs in order to guarantee that the number of possible products of the SPL do not decrease as consequence of refactoring (this is what they called guarantee *configurability improvement*).

The refactoring of FMs plays a key role in approaches such as Feature Oriented Programming (FOP) [6] in which products are generated automatically by composing programs specified declaratively in terms of features. Such features are modelled in a FM which is used as the starting point for the generation of products. In this context, the refactoring of programs could be done ideally by refactoring only its associated FM. For that purpose, multiple challenges must be overcome at the FM level and the code level. In this paper we contribute at the model level by giving a first step toward automated tool support for FM refactoring.

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web–Factories (TIN2006-00472).

SPL refactoring requires a special treatment in order to avoid loosing configuration alternatives. For that purpose, two proposals has been suggested in order to guarantee configurability improvement [1]: *i*) Analyzing the FMs to ensure that the final FM obtained after refactoring preserves all the configurations alternatives of the initial one and *ii*) Using explicit and proved FM refactorings to guarantee that the transformations do not alter negatively the capacity of configuration of the SPL.

In order to provide automated tool support for FM refactoring we studied the previously mentioned proposals and analyzed how it could be used for automating the refactoring process. Next, we detail our main conclusions:

- Our experience in the field of the automated analysis of FMs [8] and the available analyzing tools [10] and performance tests [9] let us to consider automated analysis of FMs as a suitable mechanism to check configurability improvement in small and medium size FMs. For larger FMs the computational complexity of the analysis operations needed could result excessive. However, this approach only can be used to check the configurability of a FM automatically but not for automating the refactoring transformations in any way.

- The second option pointed by Alves *et al.* and detailed in Section 3 refers to the possibility of using a catalog of pattern-based FM refactoring. FM refactorings are a set of visual pattern-based rules representing the possible transformations that can be performed on a FM without decreasing it configurability. In contrast with the previous proposal, this one does not impose any limitation in the size of the FM and could be performed automatically using any of the existing approaches for model transformations [14].

Graph Transformations are a very mature approach used since 30 years ago for the generation, manipulation, recognition and evaluation of graphs [23]. Most of visual languages can be interpreted as a type of graph (directed, labelled, etc.). This makes graph grammar to be a natural and intuitive way for transforming models. In contrast with other model transformation approaches [14], graph transformations are defined in a visual way and are provided with a set of tested tools to define, execute and test transformations. All these characteristic make graph transformations to be recognized as a suitable technology and associated formalism for model refactoring [11, 20].

In this paper, we propose to provide tool support for FM refactoring using model transformations. In particular, we propose implementing the catalog provided by Alves *et al.* using graph transformations. In addition, we details how our proposal could be integrated in the FAMA plug-in and clarify it with an example.

The remainder of this paper is organized as follow: in Section 2 the main concepts of SPL, FMs and graph transformations are presented. An introduction to the characteristics of refactoring in the context of SPL is introduced in Section 3. In Section 4 we present our proposal by giving details about how implementing the refactoring of FMs using graph transformations and how it could be integrated in the FAMA plug-in. Finally we describe our future work and summarize our conclusions in Sections 5 and 6 respectively.

2 Preliminaries

2.1 Software Product Lines and Feature Models

Software Product line (SPL) development [12] is an approach to develop families of software systems in a systematic way. Software reuse and quality are its main goals. A SPL can be defined as a set of software products that share a common set of features. Widely known examples are cars or mobile phones product lines. Feature models [18] are a key artefact for variability management in the context of SPL. A FM is a compact representation of all the possible products of a SPL. Furthermore, it is commonly accepted that FMs can be used in different stages of an SPL effort in order to produce other assets such as requirements documents [16, 17], architecture definition [22] or pieces of code [13, 7].

FMs are represented visually by means of feature diagrams. Roughly speaking, a feature diagram is a tree-like structure in which nodes represent features and connections illustrate the relations between them. The root feature identifies the SPL. The relationships between a parent feature and its child features can be divided in:

- *Mandatory.* If a child feature is mandatory, it is included in all products in which its parent feature appears.
- *Optional.* If a child feature is defined as optional it can be optionally included in all products in which its parent feature appears.
- *Alternative.* A set of child features are defined as alternative if only one feature can be selected when its parent feature is part of the product.
- *Or-Relation.* A set of child features are said to have an or-relation with their parent when one or more of them can be included in the products in which its parent feature appears.

Notice that a child feature can only appear in a product if its parent feature does. The root feature is a part of all the products within the SPL. In addition to the parental relationships between features, a feature diagram can also contain cross-tree constraints between couples of features. These are typically of the form:

- *Requires.* If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product.
- *Excludes.* If a feature A excludes a feature B, both features can not be part of the same product.

Figure 1 illustrates the visual notation of feature diagrams. A widely used example of a FM from the automotive industry is shown in Figure 2. The simplified sample diagram illustrates how features are used to specify and build software for configurable cars. The software loaded in the car control system is determined by the car's features. The sample diagram shows that every car has a body, a transmission and an engine whereas the cruise control is an optional feature. In a similar way, the types of transmission are specified by means of an alternative relationship indicating that cars can have automatic or manual transmission but not both of them. On the other hand, an or-relation is used to express that a car can have an electric engine, a petrol engine or both of them.

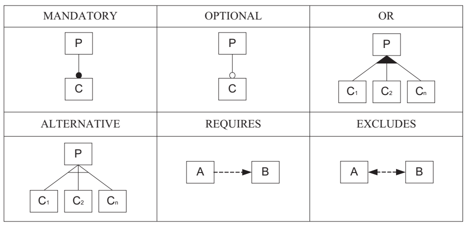


Figure 1: Feature diagram's visual notation

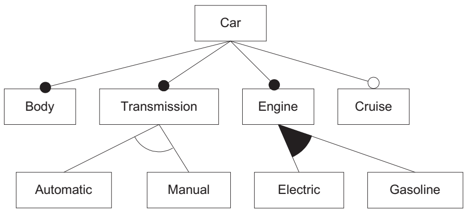


Figure 2: A feature model

The number of different possible combinations of features in a FM represent the *configurability* of the associated SPL. Thus, the configurability of a SPL establishes the set of different products that a company can offer to its customers. Such configurability can logically increase or decrease according to the modifications performed on the FM. Hence, for instance, in the FM showed in Figure 2 the number of different possible configurations is 12. However, if we convert the "cruise" feature to mandatory the number of products decrease to 6 meanwhile adding the option of having a manual and automatic transmission at the same time rise the number of possible configurations to 18.

2.2 Graph Grammars and Graph Transformations

Graph Grammars are a very mature approach used since 30 years ago for the generation, manipulation, recognition and evaluation of graphs [23]. Since then, graph grammars has been studied and applied in a variety of different domains such as pattern recognition, syntax definition of visual languages, specification of abstract data types, model refactoring, description of software architectures, etc. This development is documented in several surveys, tutorials and technical reports [2, 3, 4, 19, 20].

Graph grammars can be considered as the application of the classic Chomsky's string grammars concepts to the domains of graphs. Hence, a graph grammar is composed by an initial graph, a set of terminal labels and a set of transformation rules (sometime also called graph productions). A transformation rule is composed mainly by a source graph or Left Hand Side (LHS) and a target graph or Right Hand Side (RHS). The application of a transformation rule to a so-called host graph, also called direct derivation, consists on looking for a match morphism between the LHS and the host graph. If such morphism is found, the occurrence of the LHS in the graph is replaced by the RHS of such rule. Thus, each rule application transforms a graph by replacing a part of it by another graph. The set of all graphs labelled with terminal symbols that can be derived from the initial graph by applying the set of transformation rules iteratively is the language specified by the graph grammar.

The application of transformation rules to a given graph is called *Graph Transformations*. Graph transformations are usually used as a general rule-based mechanism to manipulate graphs. Most of visual modelling languages can be interpreted as a type of graph (directed, labelled, attributed, etc.). This make graph grammars and graph transformations to be a natural and intuitive way to define the syntax of visual languages [3], performing pattern-based visual model transformation [19] or model refactoring [20].

There exists a variety of tool for the definition of graph grammars and the application of graph transformations. Fujaba [1] and the AGG System[2] are two of the most popular general-purpose graph transformation tools within the research community. Furthermore, other specific tools such as GReAT[3] or VIATRA2[4] are also starting to emerge as a consequence of the increasing popularity of model driven development based on graph transformations.

There exist a wide variety of graph transformation approaches depending on the type of graphs used (attributed, hypergraph, directed, etc.) and how the transformation rules are applied. A deep study of them is out of the scope of this paper. A more detailed introduction to graph grammars and graph transformations can be found in [23].

3 Refactoring Software Product Lines

The term *refactoring* was first introduced by Opdyke [21] in the context of object oriented programming as behaviour-preserving program restructuring operations to support the design, evolution and reuse of object-oriented application frameworks. A more general definition is given by M. Fowler [15] who describe it as *"the process of changing software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure"*. However, such definition does not consider the special characteristics of SPL in which programs are frequently derived from a FM representing all the configuration variants in a SPL. Hence, it is recognized that in a SPL environment not only program should be refactored but also FMs [1]. In this context, the refactoring of programs and FMs can have

[1]http://wwwcs.uni-paderborn.de/cs/fujaba/
[2]http://tfs.cs.tu-berlin.de/agg/
[3]http://www.escherinstitute.org/Plone/tools
[4]http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2

the highly negative effect of reducing the configurability of the SPL.

In order to cover the intrinsic characteristics of SPL V. Alves *et al.* [1] propose a specific refactoring definition for this context: *"SPL refactoring is a change made to the structure of a SPL in order to improve (mantain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behaviour of its original products"*.

Alves *et al.* also identify two types of programs refactoring in the context of SPL:

1. Transforming a SPL into another one in which the behaviour is preserved and the structured enhanced in some way.

2. Merging multiple programs into a new product line in which the configuration alternatives of the separated programs are joined in the final SPL.

In both cases, the problem of loosing configuration alternatives could happen. In order to guarantee configurability improvement when refactoring a SPL, Alves *et al.* propose a catalog of sound *FM refactorings*. Such FM refactorings represent the possible transformations that can be performed on a FM without reducing its configurability. The soundness of such refactorings was proved using a formal semantic in a theorem prover.

A FM refactoring consists of two templates or patterns of FMs: the left-hand (LHS) and the right-hand (RHS). The LHS and RHS templates of a FM refactoring represent respectively the state of a part of a FM before and after applying some kind of refactoring on it. The application of a refactoring to a FM consists on replacing all the occurrences of the LHS by the RHS. The values of the variables in both, LHS and RHS, must match in order to perform the refactoring. Figure 3 and Figure 4 show two of the FM refactorings proposed by Alves *et al.*. The fragments of FMs placed on the left and right of the arrows represent the LHS and RHS patterns respectively. Figure 3 depicts a FM refactoring in which an optional feature is removed and added to an or relation placed under the same parent feature. On the other hand, Figure 4 shows a FM refactoring in which the FMs of two different programs are merged into a single product line.

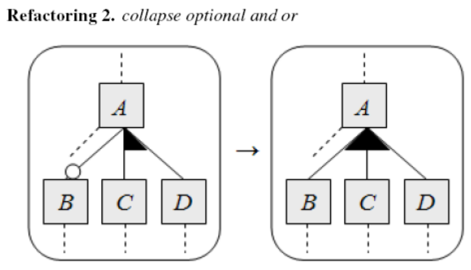


Figure 3: An example of FM refactoring (extracted from [1]).

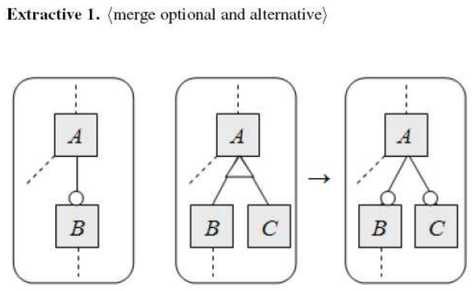


Figure 4: A FM refactoring merging two existing program in a single product line (extracted from [1]).

4 Our Proposal

In this section we introduce our proposal. Firstly, we detail how graph transformations can be used as a suitable technology and formalism to perform FM refactoring automatically. Secondly, we concrete our contribution by explaining how graph transformation could be integrated in the FAMA plug-in in order to offer automated support for FM refactoring. Finally, we clarify our proposal by means of an example.

4.1 Automating Feature Model Refactoring using Graph Transformations

In this paper we propose using model transformations as an appropriate mechanism to provide automatic support for FM refactoring. In particular, we propose implementing the catalog of FM refactorings provided by Alves *et al.* using graph transformations.

Graph transformations are recognized as a suitable technology and formalism for the specification and application of model refactorings. In [20] Mens introduces the main concepts of graph transformations theory and tools and show how using it for refactoring UML class and statechart diagrams. A more general contribution is proposed by Biermann *et al.* [11] who use graph transformations to apply refactoring on EMF (Eclipse Modeling Framework) models. Hence, as documented in the literature, the reasons to select graph transformations as a suitable approach for model refactoring are manifold:

- Graph transformations are a natural and intuitive way of performing pattern-based visual model transformations.
- The maturity of graph transformations has provided it with a solid theoretical foundation in form of useful properties. Hence, for instance, the "invertability" property details under what conditions a transformation rule can be inverted.
- There are available tools to design, execute and test the transformations rules. Example of them are the previously mentioned AGG, Fujaba, GReAT or VIATRA2.

In order to test our proposal we implemented it in one of the most popular tool within the graph grammar community: *The Attributed Graph Grammar System (AGG)*. Roughly speaking, AGG is a free Java graphical tool for editing and transforming graphs by means of graph transformations. The AGG System is a prototype implementation of the algebraic approach to graph transformation supporting *Contextual Layered Graph Grammars* (CLGG). In CLGGs the set of productions is classified into ordered layers. To transform a graph, productions are applied layer by layer from layer 0 to layer N, cyclically if needed, until none of the productions can be applied. AGG provides a flexible graph editor and a useful component to apply user-selected productions to a given graph. In addition, the AGG system can be used as a general purpose graph transformation engine in any dedicated Java applications employing graph transformation methods. All this reasons made us to select AGG as a suitable tool to implement our proposal.

AGG graph transformation rules consist on three main parts: a left-hand side graph (LHS), a right-hand side graph (RHS) and a set of Negative Application Conditions (NAC). NACs are graph-based patterns representing under what conditions the rule will not be applied. Thus, the mapping from the FM refactorings to graph transformations is straightforward since both approach use pattern-based concepts. Figure 5 shows a screenshot of the AGG's visual editor displaying the transformation rule associated to the FM refactoring showed in Figure 3. The mapping consists on translating the LHS and RHS patterns of the FM refactoring to the LHS and RHS graphs of the AGG transformation rule respectively. No NACs are needed in this case. As illustrated in the figure we have used an specific visual syntax due to the restrictions imposed by the visual editor. In particular, we have used solid arrows for the connections in the or relation and a dashed arrow for the optional one.

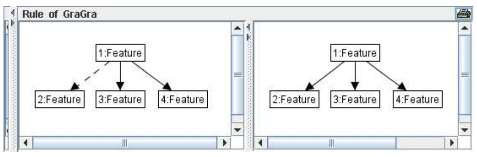


Figure 5: AGG transformation rule

4.2 Tooling Automated Support

In this section we concrete our proposal by detailing how we plan to integrate the automatic support for FM refactoring in the FAMA plug-in.

FAMA (FeAture Model Analyzer) [10] is a multiparadigm framework for the edition and automated analysis of FMs. It integrates different logic paradigms and solvers in order to perform the analysis operations. FAMA is implemented as an extensible Eclipse plug-in making it possible to integrate new solver and analysis operations as required. In addition, it supports the import and export of FMs in XML/XMI format facilitating the interoperability with other related tools. FAMA is especially appropriate for the edition and manipulation of FMs in the early stages of SPL development. However, it could be extended to support others SPL development tasks such as design modelling or feature composition at the code level by means of feature–oriented programming tools.

According to the kind of program refactorings identified in SPL we distinguish two types of automatic support to be provided:

- If the refactoring is applied to a single program and its FM, the possible transformations to apply depend exclusively on the kind of refactoring to be performed. In this case, we propose a semi-automatic solution implemented by means of a context-sensitive help wizard. Such wizard could suggest to the user the possible refactoring transformations according to the features involved on it.
- If the refactoring consists on merging the FMs of multiple programs into a single one, we consider it can be automatized completely since all the information required by the transformation system is available. In this case, we propose a transformation wizard for merging both FMs into a single one.

In both cases, we plan to use graph transformations to perform model transformations and the AGG System as suitable graph transformation engine for being integrated in FAMA.

4.3 An Example

A software company specialized in mobile phone control systems provides a wide variety of related products to its customers. The company has noticed that all its products shares a common set of features and it has decided to adopt a SPL strategy in order to reduce development costs and time-to-market. As a first step, the software architect has decided to design the FMs of two of its main programs. Figure 6(a) and Figure 6(b) depict the resultant FMs. Notice that the alternatives of configuration in both cases are minimum.

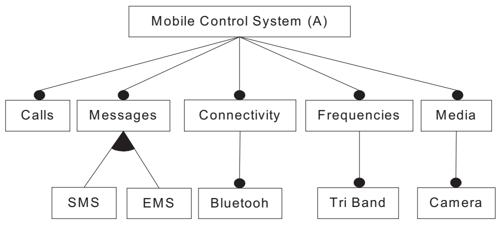
After the "featurization", the software architect has decide to refactor one of the programs and its associated FM in order to increase the configurability of the future SPL. Figure 7 depicts an interface prototype of how our proposal could be integrated in the customized tree view editor of FAMA. The figure shows how a context-sensitive menu could be used to suggest the possible FM refactorings according to the selected feature. In this case, the software architect has decided to convert the mandatory feature "wifi" to optional.

As a natural step during the adoption of a SPL strategy the software architect feels the need of merging both programs and its associated FMs into a single product line. In this case, our proposal could be also used to merge automatically both FMs into a single one preserving configurability. Figure 6(c) shows the resultant FM obtained after merging the FMs of both programs into a single product line.

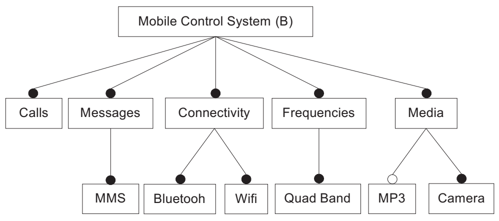
5 Future Work

Multiples challenges must be overcome to provide an efficient automated tool support for FM refactoring based on model transformations. In particular, we have identified three promising focus of research in such direction:

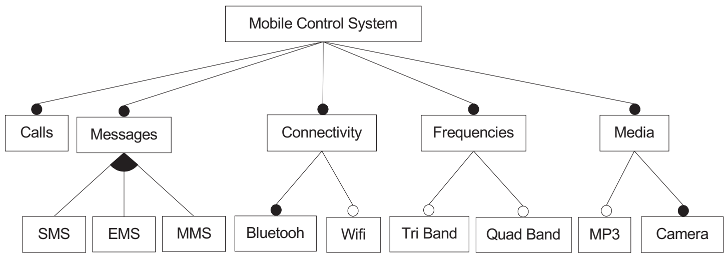
- Alves *et al.* mantain that refactorings consisting on merging multiple FMs into



(a) FM of a mobile phone control system (A)



(b) FM of a mobile phone control system (B)



(c) Resultant FM after merging the FMs in 6(a) 6(b)

Figure 6: SPL refactoring. Merging two programs into a single one

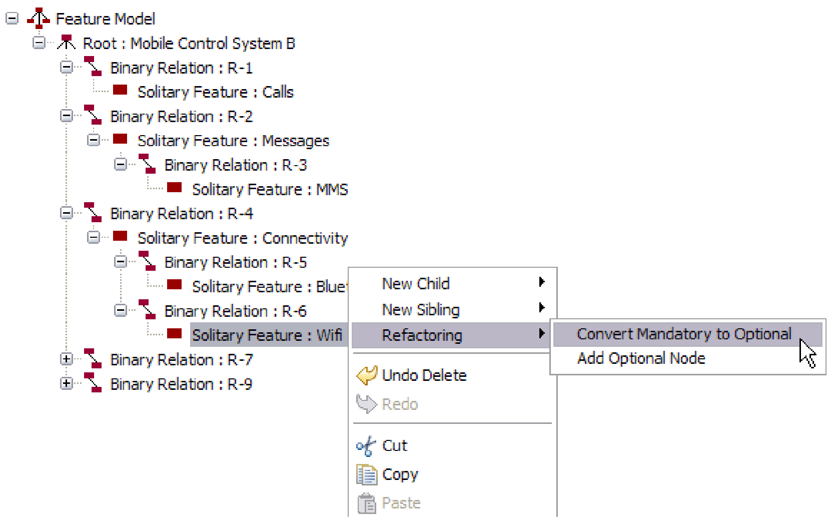


Figure 7: GUI prototype of the refactoring operations in FAMA

a single one can be performed by applying a sequence of single FM refactorings to the original FMs separately. However, we consider that an specific catalog of FM merging refactorings would be a promising approach in order to automate completely the merging process.

- FMs based on multiplicities [13] (also called cardinality-based FMs) are a widely used extension of the traditional notation presented in this paper. However, no specific FM refactorings have been proposed for them. Therefore, we consider that a catalog of cardinality-based FMs refactorings would be an interesting approach for the SPL community.

- The context-sensitive application of a transformation rule to a FM is not a trivial task. The possible mappings between the selected features and the patterns of the transformation rules must be analyzed in order to offer a good performance. Hence, we consider that the selection of a suitable mapping strategy is an important issue and we plan to work in such direction too.

6 Conclusions

In this paper we present a proposal for automated support for FM refactoring based on model transformations. In particular, we propose implementing the catalog of FM refactorings provided by Alves *et al.* using graph transformations. As part of our proposal we first introduced the mapping from a FM refactoring to a graph transformation rule in the AGG system. Next, we proposed how graph transformations could be integrated in the FAMA plug-in to provide tool support for FM refactoring. Finally, we provided an example in order to make clear the interest of our contribution.

Graph transformations are a mature, natural, visual and intuitive means for manipulating visual models. It has already been applied successfully in the context of model refactoring. In contrast with other proposals, graph transformations theory provides a solid formal foundation and a set of solid tools. To the best of our knowledge, this is the first work proposing automating FM refactoring by means of model transformations.

References

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210, New York, NY, USA, 2006. ACM Press.

[2] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[3] R. Bardohl, M. Minas, A. Schurr, and G. Taentzer. *Application of graph transformation to visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation, volume II: Applications, Languages and Tools.* World Scientific, 1999.

[4] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 402–429, London, UK, 2002. Springer-Verlag.

[5] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.

[6] D. Batory. A tutorial on feature oriented programming and the ahead tool suite. In

Summer school on Generative and Transformation Techniques in Software Engineering, 2005.

[7] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[9] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[10] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[11] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Emf model refactoring based on graph transformation concepts. In *In Proc. Third International Workshop on Software Evolution through Transformations (SETra'06)*, volume 3 of *Electronic Communications of the EASST*, 2006.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison–Wesley, August 2001.

[13] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addison–Wesley, may 2000. ISBN 0–201–30977–7.

[14] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[15] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, Reading/Massachusetts, 1999.

[16] G. Halmans and K. Pohl. Communicating the variability of a software–product family to customers. *Journal on Software and Systems Modeling*, 2(1):15–36, 2003.

[17] S. Jarzabek, W. Ong, and H. Zhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.

[18] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[19] T. Mens, P. van Gorp, G. Karsai, and D. Varró. Applying a model transformation taxonomy to graph transformation technology. In G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, ENTCS, 2005. In press.

[20] Tom Mens. On the use of graph transformations for model refactoring. *Generative and Transformational Techniques in Software Engineering. LNCS*, 4143:219–257, 2006.

[21] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.

[22] J. Peña, M. Hinchey, A. Ruiz-Cortés, and P. Trinidad. Building the core architecture of a multiagent system product line: With an example from a future nasa mission. In *7th International Workshop on Agent Oriented Software Engineering.* LNCS, 2006.

[23] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.