Automated Merging of Feature Models using Graph Transformations *

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

University of Seville, Spain
{sergiosegura,benavides,aruiz,ptrinidad} AT us.es

Abstract. Feature Models (FMs) are a key artifact for variability and commonality management in Software Product Lines (SPLs). In this context, the merging of FMs is being recognized as an important operation to support the adoption and evolution of SPLs. However, providing automated support for merging FMs still remains an open challenge. In this paper, we propose using graph transformations as a suitable technology and associated formalism to automate the merging of FMs. In particular, we first present a catalogue of technology-independent visual rules to describe how to merge FMs. Next, we propose a prototype implementation of our catalogue using the AGG system. Finally, we show the feasibility of our proposal by means of a running example inspired by the mobile phone industry. To the best of our knowledge, this is the first approach providing automated support for merging FMs including feature attributes and cross-tree constraints.

1 Introduction

Software Product Line (SPL) engineering is an approach to developing families of software systems in a systematic way [10]. Roughly speaking, an SPL can be defined as a set of software products sharing a common set of features. A feature is defined as an increment in product functionality [5]. In this context, *Feature Models* (FMs) are commonly used to provide a compact and visual representation of all the products of an SPL in terms of features.

Typical SPL adoption strategies involve building an SPL from a set of existing software products or extending an existing SPL to include new products [17]. In both cases, the artifacts of different software systems must be combined into a single SPL. In this context, the usage of specific SPL refactoring techniques is emerging as a key practice to support the adoption and evolution of SPLs [1,18,30].

It is accepted that not only programs should be refactored in the context of an SPL but also FMs. In particular, the merging of FMs emerges as an appealing operation to support the evolution of SPLs at the model level [1,12]. In this context, different semantics for the operation of merging of FMs have been proposed in the literature [26]. For the proof of concept presented in this paper, we consider the merging of FMs as an operation that takes as input a set of FMs and returns a new FM representing, as a

minimum, the same set of products than the input FMs. Once this FM representing all products is generated, it may be used as a starting point for driving future development.

Graph Transformations are a very mature approach, having been used for 30 years for the generation, manipulation, recognition and evaluation of graphs [25]. Most visual languages can be interpreted as a type of graph (directed, labeled, etc.). This makes graph transformations a natural and intuitive way for transforming models [11,13,21]. Graph transformations are defined in a visual way and are provided with a set of tested tools to define, execute and test transformations. Additionally, graph transformation theory provides a solid formal foundation enabling the checking of interesting formal properties such as confluence, sequential and parallel (in)dependence, etc. [15]. All these characteristics make graph transformations a suitable technology and associated formalism for model refactoring [20,23] and software merging [19,31].

In previous work [27] we detailed our intention of providing automated tool support for FM refactoring using graph transformations. In this paper we present our first results in that direction. In particular, the contribution of this paper is twofold:

– We propose a catalogue of 30 visual rules to merge FMs. In contrast to existing proposals, our catalogue includes rules to describe how to merge FMs including feature attributes and cross-tree constraints.
– We propose using graph transformations as a suitable technology to automate the merging of FMs. In order to show the feasibility of our proposal, we present a prototype implementation of our catalogue of rules using the AGG system [29]. To the best of our knowledge, this is the first approach providing automated support for merging FMs including feature attributes and cross-tree constraints.

The remainder of the paper is structured as follows: in Section 2, the main concepts of feature models and graph transformations are introduced. Our proposal is presented in Section 3. In Section 4, we survey related work. We describe the main challenges remaining for our future work in Section 5. Finally, we summarize our main conclusions in Section 6.

2 Preliminaries

2.1 Feature Models

Feature Models (FM) [16] are used to model sets of software systems in terms of features and relations among them (see Figure 1). A feature can be defined as an increment in product functionality [5]. FMs are commonly used as a compact representation of all the products of an SPL in terms of features.

A FM is visually represented as a tree-like structure in which nodes represent features, and connections illustrate the relationships between them. Figure 1 depicts a simplified example FM inspired by the mobile phone industry. The model illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. The root feature identifies the SPL. The relationships between a parent feature and its child features can be divided into:
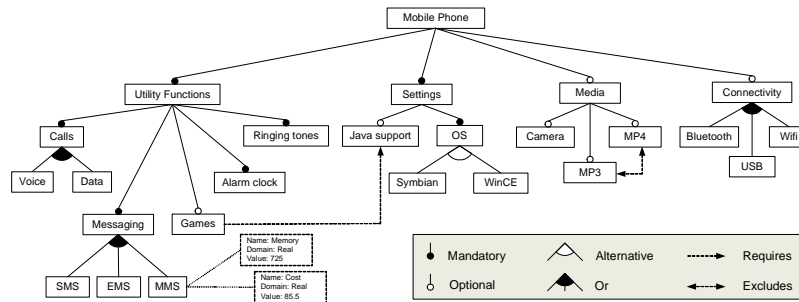
Fig. 1: A sample feature model

- *Mandatory*. If a child feature is mandatory, it is included in all products in which its parent feature appears. Hence, for instance, according to the sample model, all mobile phones must provide support for *ringing tones*.
- *Optional*. If a child feature is defined as optional, it can be optionally included in all products in which its parent feature appears. For instance, the sample feature model defines *games* as an optional feature.
- *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. As an example, according to the model, a mobile phone will use a *Symbian* or a *WinCE* operating system but not both in the same 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. Hence, for instance, according to the sample model, a mobile phone can provide connectivity support for *bluetooth*, *USB*, *wifi* or any combination of the three.

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 FM can also contain cross-tree constraints between 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. Hence, for instance, in the example shown, mobile phones including *games* require *Java support*.
- *Excludes*. If a feature A excludes a feature B, both features cannot be part of the same product. As an example, the SPL represented in Figure 1 removes the possibility of offering support for *MP3* and *MP4* formats in the same product.

Feature Models were first introduced as a part of the Feature-Oriented Domain Analysis method (FODA) by Kang back in 1990 [16]. Since then, multiple extensions to the traditional notation have been proposed in order to increase its expressiveness [7]. In this context, some well known extensions are the so-called *Extended Feature Models* [4,5,6]. Roughly speaking, extended FMs propose adding extra-functional information to the features using attributes. There is no consensus on a notation to define

attributes. However, most proposals agree that an attribute should consist at least of a *name*, a *domain* and a *value*. For instance, the feature 'MMS' in Figure 1 includes some feature attributes using the notation proposed by Benavides *et al.* in [6]. As illustrated, attributes can be used to specify extra-functional information such as cost, speed or RAM memory required to support the feature.

2.2 Graph Transformations

Graph Grammars are a mature approach for the generation, manipulation, recognition and evaluation of graphs [25]. Graph grammars have been studied and applied in a variety of different domains such as pattern recognition, syntax definition of visual languages, model transformations, description of software architectures, etc. This development is documented in several surveys, tutorials and technical reports [3,13,20,21,25].

Graph grammars can be considered as the application of the classic string grammar concepts to the domain of graphs. Hence, a graph grammar is composed of an initial graph, a set of terminal labels and a set of transformation rules (sometimes also called graph productions). A transformation rule is composed mainly of 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 of looking for an occurrence of the LHS graph in the host graph. If this match is found, the occurrence of the LHS in the graph is replaced by the RHS of the given 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 Transformation*. Graph transformations are usually used as a general rule-based mechanism to manipulate graphs. Most visual modelling languages can be interpreted as a type of graph (directed, labelled, attributed, etc.). This makes graph transformations recognized as a suitable technology for the specification and application of model transformations [11,13,21], model refactoring [20,23] and software merging [19,31]. Hence, as documented in the literature, the reasons to select graph transformations as a suitable approach for the transformation and refactoring of visual models 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 the form of useful properties [15,22]. Hence, for instance, the properties of *sequential* and *parallel dependence* are used to detect the set of transformation rules that must be applied in a given sequence and the set of rules that can be applied in parallel.
– There is a variety of mature tools to define, execute and test transformations rules. Fujaba [1] and the AGG System[2] are two of the most popular general-purpose graph

---

[1] http://wwwcs.uni-paderborn.de/cs/fujaba/

[2] http://tfs.cs.tu-berlin.de/agg/

transformation tools within the research community. Nevertheless, other tools such as GReAT[3], VIATRA2[4] or GROOVE[5] are also starting to emerge as a consequence of the increasing popularity of graph transformations in the model-driven development domain.

3 Our proposal

In this section we present our proposal. In particular, we first propose a catalogue of visual rules describing how to merge FMs. Then, we present a prototype implementation of the proposed catalogue using graph transformations and the AGG system. Finally, we clarify our contribution by means of an example inspired by the mobile phone industry.

3.1 Catalogue of rules

In Appendix A (see page 15), we present a catalogue of 30 visual, technology-independent rules to describe how to merge FMs. More specifically, our catalogue of *merge rules* describes how to build a FM including all the products represented by two given FMs.

Each merge rule consists of two input patterns of the FMs to be merged (preconditions) and an output pattern of the new FM generated as a result of the merging (postconditions). The rules can be iteratively applied by looking for matches in the input patterns on the two FMs to be merged. A match is an assignment of the variables of the patterns to concrete values. The elements not mentioned in any of the patterns remain unchanged by default. The result of the merging is defined as a new FM including all the products represented by the merged FMs. This resulting FM could be later used as an input model in any other merging operation.

As previously mentioned in Section 1, the merging of FMs makes sense especially when dealing with a set of related products in an SPL domain. Therefore, in order to make this first proposal possible, we make a double assumption:

– Input FMs represent related products using a common catalogue of features. For the sake of simplicity, we assume that features with the same name[6] refer to the same feature and consequently to the same software artifacts.
– The parental relationship between features is equal in all the FMs. That is, a feature must have the same parent feature in all the models in which it appears.

Figure 2 depicts one of the merge rules defined in our catalogue. The input and output patterns of the rule are placed on the left and right side of the arrow respectively. The sample rule illustrates the case in which a feature is defined as a child of an or-relationship and as a mandatory feature in both inputs FMs respectively. As a result of the merging operation, the feature is included in an or-relationship in the resulting FM preserving the configurability options (products) and the grouping structure.

---

[3] http://www.escherinstitute.org/Plone/tools

[4] http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2

[5] http://groove.sf.net

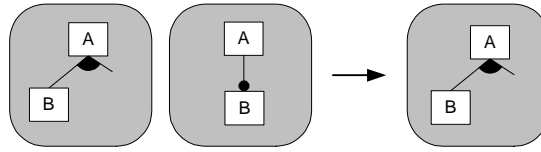[6] A name could be a string, an identifier, a signature, etc.

Fig. 2: A sample merge rule

Providing basic support for merging extended FMs is also a part of our contribution. To this aim, we also define rules describing how to merge feature attributes. As an example, Figure 3 illustrates the case in which a given feature has an attribute with the same name in both input FMs. More specifically, both attributes have the same name and value but different domains. As a result of the merging operation, an attribute with the same name and value is created in the resulting model. The domain of the new attribute consists of the union of the domains of the merged attributes. This way we guarantee that the attribute can take the same values as in the input FMs.
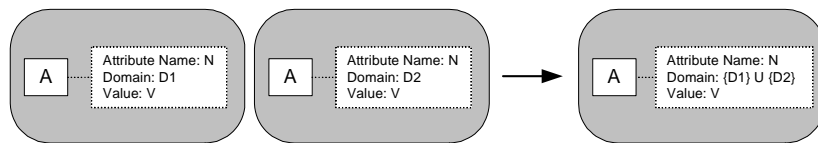

Fig. 3: Merging feature attributes

3.2 Correctness of the catalogue

We consider our catalogue of merge rules to be correct if it guarantees that the set of products represented by the resulting FM includes, as a minimum, the set of products represented by the merged FMs. Performing an exhaustive check of the correctness and completeness of the catalogue of rules is out of the scope of this paper. However, in order to perform a preliminary validation of the catalogue, we used FAMA[7] [8], a framework for the edition and automated analysis of FMs. Next, we describe the main steps we followed to test each merge rule:

1. We modelled two example input FMs that matched the input patterns of the merge rule to be validated.
2. We used FAMA to extract the set of products represented by the example input FMs.
3. We created a new FM simulating the application of the given merge rule to the example input FMs.

---

[7] http://www.isa.us.es/fama

4. We used FAMA again to extract the set of products represented by the new FM.
5. We finally checked that the set of products represented by the input FMs were included in the set of products represented by the output FM.

3.3 Implementation using graph transformations

In this section we propose using model transformations as an appropriate mechanism to provide automated tool support for merging FMs. In particular, we present a prototype implementation of our catalogue of merge rules for merging FMs using graph transformations.

In order to implement our proposal, we selected a popular tool within the graph grammar community: *The Attributed Graph Grammar System (AGG)*[8] [29]. AGG is a free Java graphical tool for editing and transforming graphs by means of graph transformations. AGG graphs may be typed over a type graph and attributed by Java objects and types. Rule application order may be controlled by dividing rules into layers. Due to its formal foundation, AGG offers validation support for consistency-checking of graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules [20,22] and checking of termination criteria. All these reasons made us select AGG as a suitable tool to implement our proposal.

AGG graph transformation rules consist of three parts: a left-hand side graph (LHS) and a right-hand side graph (RHS), a mapping between nodes and edges on both sides and a set of Negative Application Conditions (NACs). NACs are preconditions prohibiting certain object structures on the graph from being transformed. Figure 4 shows a screenshot of the AGG GUI. On the left hand side, a tree view displays the working graph and the rules of the proposed grammar. In the upper central area, the NAC (if any) and the LHS and RHS graphs of the selected rule are displayed. Finally, the central area is reserved for the host graph.

Merge rules and graph transformation rules are based on very similar concepts. In particular, both approaches use visual patterns to describe modifications in the structure of a model/graph in terms of pre- and postconditions. Hence, in order to implement our proposal, we mapped our catalogue of merge rules into AGG rules. Next, we outline the main steps we followed to implement our proposal in AGG:

1. Firstly, we defined a set of typed nodes and edges in order to represent FM as graphs. Hence, for instance, we defined a feature as a type of node and an optional relationship as a type of edge. Additionally, we set different visual layouts for the different types of nodes and edges in order to make graphs easily comprehensible. As an example, we used solid and dashed edges to represent mandatory and optional relationships respectively.
2. Next, we created an attributed type graph. From an intuitive point of view, a type graph may be considered as a meta-graph expressing the well-formedless rules that must hold for all graphs. AGG type graphs may include abstract nodes (i.e. not instantiable), node type inheritance and UML-like multiplicities. Graphs obtained as a result of a transformation rule are automatically checked for compliance with
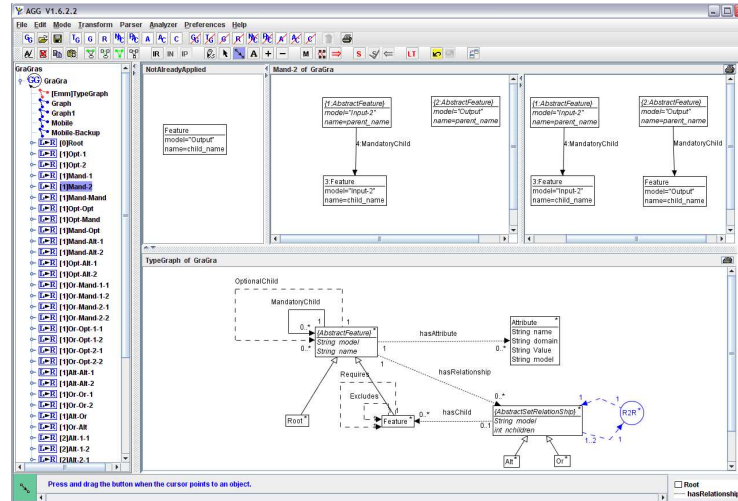
---

[8] Version 1.6.2.2

Fig. 4: The AGG System

the type graph. This way, consistency-checking is automatically performed during the merging process. In the current version of our prototype, the type graph was based on a simplified version of the meta-model for attributed FMs presented in [9].

3. The following step was mapping each merge rule into one or more AGG rules. To this aim, we implemented the input and output patterns of the merge rules as the LHS and RHS graphs of the graph transformation rule respectively.

4. In addition to the LHS and RHS graphs, we finally defined additional NACs to restrict when rules can be applied. Hence, for instance, typical NACs avoid the execution of a transformation rule more than once.

As an example, Figure 5 shows a screenshot of the AGG rules used to implement the merge rule presented in Figure 2. From left to right, the NAC, LHS and RHS graphs of each rule are presented. As illustrated, different typed nodes are used to represent features and or relationships. In a similar way, different types of edges are used to represent the relationships between features. Additionally, both nodes and edges are provided with attributes which are used to set properties such as the name of features or the kind of FM a node/edge is representing (i.e. input or output).

In addition to the elements of the model, we also used some helper structures and attributes to implement our proposal. This is a common practice when implementing graph transformations [13]. For instance, we used auxiliary nodes (visually represented as circles) to maintain traceability between the or relationships in the input and output graphs. In a similar way, we used attributes to define helpful properties as the number of children of the or relationships (see Figure 5).

Both AGG rules execute the same merging operation on the models but assume a different starting situation. On the one hand, transformation rule 1 is executed when the

or-relation has not been created on the output FM yet. In this case, the application of the rule implies the creation of the or-relation. On the other hand, transformation rule 2 illustrates the case in which the or-relation has been previously created as a result of a previous transformation rule. In both cases, a NAC is used to guarantee that the rule is applied to the same elements only once.
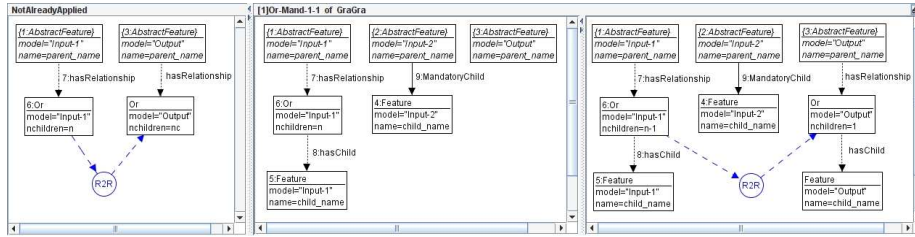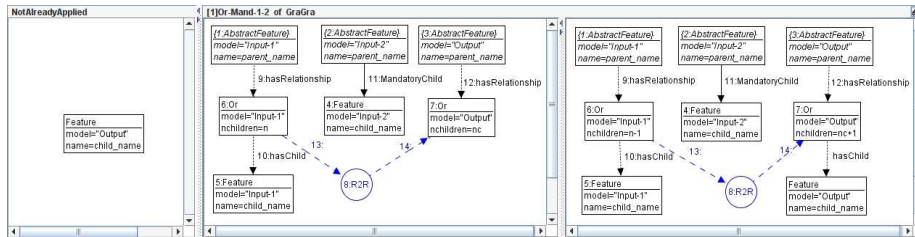

Fig. 5: AGG rules

AGG works exclusively with the graphs created using its editor and not with external models. Thus, input models must be represented as AGG graphs before applying transformations. In a similar way, once the transformation is performed, the obtained graph must be translated to the target model. The automated translation of a FM to an AGG graph and vice versa is out of the scope of this paper. However, since AGG uses XML to store the graphs, we consider XSL transformations could be used as a suitable strategy for the translations model-to-graph and graph-to-model in the context of AGG.

3.4 Overview and running example

A software company specialized in mobile phone control systems provides two main families of products to its customers. The company has noticed that both families of products share a wide common set of features, and it has decided to combine all of them into an SPL in order to reduce development costs and time-to-market.

As a first step, the software architect has decided to design the FM of the SPL as a starting point for driving future development. However, the high number and complexity of existing products make the design of this FM a time-consuming and error-prone

activity. As a result of this, the software architect decides to use an automated approach like the one presented in this paper in order to improve the efficiency and reliability of the process. Next, we summarize the main steps he/she should follow to apply or re-use our proposal:

1. Design the catalogue of merge rules to be used. Notice that different application domains could require different merging criteria. In a similar way, other extensions to the traditional notation of FMs could require a different set of merge rules. In our example, the software architect selects our catalogue of rules as a suitable approach to merge extended FMs.
2. Check the correctness of the catalogue. We have proposed using automated analysis of FMs by means of FAMA. However, other alternatives such as using theorem provers may be also feasible [1].
3. Implement the catalogue of rules. We propose using graph transformations and the AGG system as a suitable approach. However, we consider other graph transformation engines could also be used for this purpose. In the context of our example, the software architect decides to use our implementation of the catalogue in AGG.
4. Design a common catalogue of features and use it to model the FMs to be merged. Figure 6 depicts the simplified FMs of the two families of products of the company.
5. Execute the merging process. Figure 7 illustrates a screenshot of the FM generated in AGG as a result of the automated merging process. Notice that the input FMs and some of the attributes are not included due to space constraints.
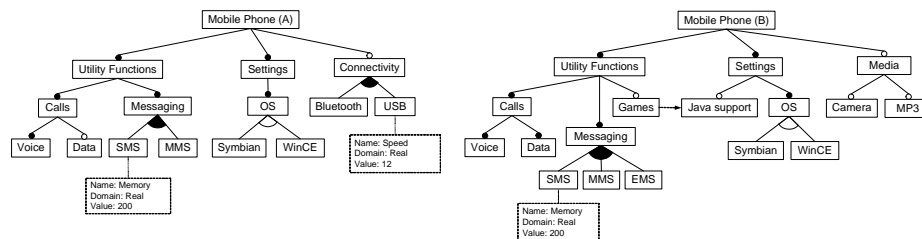
Fig. 6: Feature models to be merged

Once the resulting FM is generated, it could be automatically analysed to extract helpful information such as the number of potential products, commonality of features, set of features of minimum cost, etc [8]. This information could be later used to make relevant design decisions such as selecting the set of features that should be part of the core architecture of the SPL [24].

4 Related Work

This work is partially inspired by the proposal of Alves *et al.* [1], in which they motivate the need for refactoring FMs. In their work, the authors propose a catalogue of

Fig. 7: Automated merging of feature models in AGG

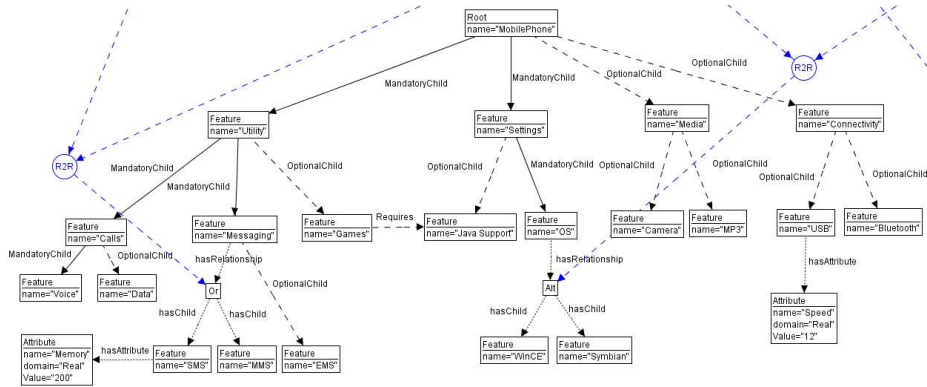refactoring rules describing the refactoring operations that can be performed on single FMs. They also motivate the need for merging FMs (what they call *bidirectional refactorings*). Additionally, the authors propose using the automated analysis of FMs, using Alloy as a suitable mechanism, to check the correctness of the catalogue [14]. In contrast to our work, the merging of extended FMs or the automated support for FM refactoring are topics not covered by their proposal. Nevertheless, we presume that graph transformation could be also used as a suitable mechanism to implement their catalogue. This way, their proposal could be complementary to ours for providing a complete tool support for FM refactoring.

Schobbens *et al.* [26] survey feature diagrams variants and generalize the various syntaxes through a generic artifact called *Free Feature Diagrams* (FFD). In their work, the authors identify and define three kinds of merging operations on FMs: *intersection*, *union* and *reduced product*. To the best of our knowledge, they do not provide automated support for the merging of FMs. However, we consider this a complement to our proposal, since it states clearly the semantic of the different merging operations on FMs. In this context, we presume that our proposal could be used to implement any of the identified merging operations by designing an appropriate catalogue of rules.

Czarnecki *et al.* [12] propose an algorithm to compute a FM from a given propositional formula. They point at reverse engineering and merging of FMs as some of the main applications of their approach. In contrast to our work, their algorithm does not support the merging of feature attributes and cross-tree constraints.

Another related work is proposed by Apel *et al.* [2], who present an algebra for Feature-Oriented Software Development (FOSD). As part of their approach, they introduce the so-called Feature Structure Trees (FST) as a mechanism to organize the structural elements of a feature hierarchically. In this context, the authors present a procedure for composing features based on the composition (merging) of FST using tree superimposition. Roughly speaking, tree superimposition describes how to compose trees by starting from the root and proceeding recursively. As in our work, they assume that nodes with the same name refer to the same software artifacts. Compared to our

work, they focus on single features instead of complete FMs. In addition, they do not consider cross-tree constraints or feature attributes as we explore in our proposal.

Liu *et al.* study SPL refactoring at the code level and propose what they call Feature Oriented Refactoring (FOR) [18]. They focus on providing a semi-automatic refactoring methodology to enable the decomposition of a program, usually legacy, into features. This approach complements our work, since it could be used as a suitable strategy to obtain the FMs of the legacy systems to be merged into an extractive approach.

5   Discussion and Future Work

In this paper we present our first research results toward the automated merging of FMs by using graph transformations. However, there still are many open issues that must be addressed to provide a solid tool support. In particular, we identify several challenges for our future work:

– Providing formal semantics to our approach. To this aim, we plan to define the merge operation using a formal semantic for FMs [26].
– Validating the catalogue or rules to be correct and complete according to the given semantics. We consider that the tools for the automated analysis of FMs may be helpful for that purpose. However, we also plan to study theorem provers and model checkers such as PVS[9] or Groove[10] for that aim.
– For the proof of concept performed in this paper, we deliberately made some strong assumptions (e.g. feature must have the same name). A more complex approach should allow the merging of FMs including synonym names or different attribute domains. We plan to study the works in the area of the integration of database schemas and ontologies for this aim [28].
– As we previously mentioned, some of the main advantages of using AGG are its mechanisms for consistency-checking and conflict analysis of rule applications. Exploiting massively these mechanisms and especially the critical pair analysis technique to detect conflicts between merge rules [22] is an important part of our on-going research.
– Finally, we plan to make our proposal available by integrating it into the FAMA plug-in [8].

6   Conclusions

In this paper we propose using graph transformations as a suitable technology and associated formalism to implement the merging of FMs. In particular, we first presented a catalogue of visual rules to describe how to merge FMs. In this context, we detailed how we used the FAMA plug-in for a basic validation of the catalogue. Then, we introduced a prototype implementation of our catalogue using graph transformations and the AGG system. Finally, we looked at how to apply and re-use our proposal by means of a running example inspired by the mobile phone industry. In contrast to existing proposals,

---

[9] http://pvs.csl.sri.com/
[10] http://groove.sf.net

we support the merging of FMs including feature attributes and cross-tree constraints. We also emphasize that our proposal could be extended or adapted to support other merging criteria (e.g. intersection) or FM's notations.
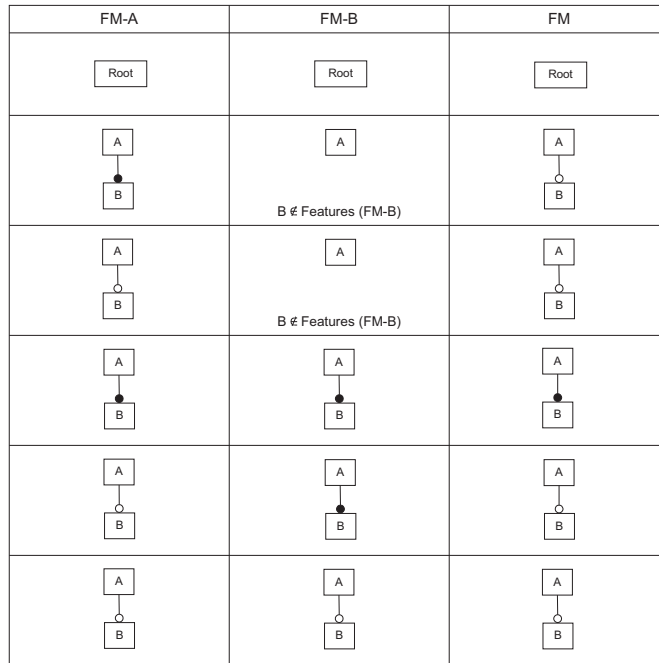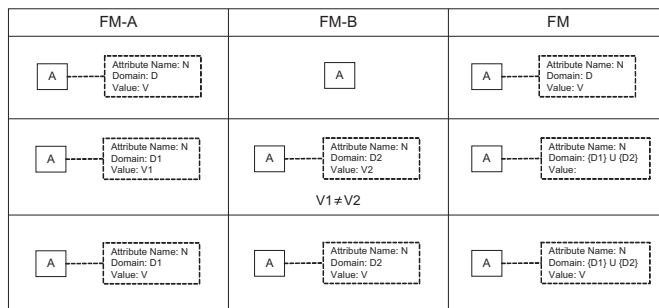
Acknowledgments

References

1. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. 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. S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. An algebra for feature-oriented software development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, Germany, July 2007.
3. L. Baresi and R. 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.
4. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.
5. D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.
6. 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.
7. D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.
8. 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.
9. D. Benavides, S. Trujillo, and P. Trinidad. On the modularization of feature models. In *First European Workshop on Model Transformation*, September 2005.
10. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.
11. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
12. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
13. K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

14. Rohit Gheyi, Tiago Massoni, and Paulo Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006.

15. R. Heckel, J. Malte Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag.

16. 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.

17. C.W. Krueger. Easing the transition to software mass customization. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, London, UK, 2002. Springer-Verlag.

18. J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM Press.

19. T. Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 127–143, London, UK, 2000. Springer-Verlag.

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

21. T. Mens, P.V. Gorp, D. Varró, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 152:143–159, 2006.

22. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.

23. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.

24. 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.

25. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

26. P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.

27. S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Toward automated refactoring of feature models using graph transformations. In Ernesto Pimentel, editor, *VII Jornadas sobre Programación y Lenguajes, PROLE 2007*, pages 275–284, Zaragoza. Spain, September 2007.

28. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics IV*, pages 146–171, 2005.

29. G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *2nd Int. Workshop on Applications of Graph Transformation*, volume 3062 of *Lecture Notes in Computer Science*. Springer, 2004.

30. S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 191–200, New York, NY, USA, 2006. ACM Press.

31. B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, New York, NY, USA, 1991. ACM.

Appendix A. Merge Rules

FM-A	FM-B	FM
Root	Root	Root
A ● B	A B ∉ Features (FM-B)	A ○ B
A ○ B	A B ∉ Features (FM-B)	A ○ B
A ● B	A ● B	A ● B
A ○ B	A ● B	A ○ B
A ○ B	A ○ B	A ○ B

(a) Root and binary relationships

FM-A	FM-B	FM
A --- Attribute Name: N / Domain: D / Value: V	A	A --- Attribute Name: N / Domain: D / Value: V
A --- Attribute Name: N / Domain: D1 / Value: V1	A --- Attribute Name: N / Domain: D2 / Value: V2 V1 ≠ V2	A --- Attribute Name: N / Domain: {D1} U {D2} / Value:
A --- Attribute Name: N / Domain: D1 / Value: V	A --- Attribute Name: N / Domain: D2 / Value: V	A --- Attribute Name: N / Domain: {D1} U {D2} / Value: V

(b) Feature attributes

FM-A	FM-B	FM
A / B	A B ∉ Features (FM-B)	A / B
A / B	A / B	A / B
A / B	A / B	A / B
A / B	A / B	A / B
A / B	A / B	A / B
A / B	A B ∉ Features (FM-B)	A / B
A / B	A / B	A / B
A / B	A / B	A / B
A / B	A / B	A / B

(c) Or-/Alternative relationships

FM-A	FM-B	FM
A ⇢ B	A,B ∉ Features (FM-B)	A ⇠⇢ B
A ⇠⇢ B	A,B ∉ Features (FM-B)	A ⇠⇢ B
A ⇢ B	A B ∉ Features (FM-B)	A B
A ⇢ B	B A ∉ Features (FM-B)	A ⇢ B
A ⇠⇢ B	A B ∉ Features (FM-B)	A ⇠⇢ B
A ⇠⇢ B	B A ∉ Features (FM-B)	A ⇠⇢ B
A ⇢ B	A B	A B
A ⇠⇢ B	A B	A B
A ⇢ B	A ⇠ B	A B
A ⇢ B	A ⇠⇢ B	A B
A ⇢ B	A ⇢ B	A ⇢ B
A ⇠⇢ B	A ⇠⇢ B	A ⇠⇢ B

(d) Cross-tree constraints