STRUCTURAL VALIDATION OF SOFTWARE PRODUCT LINE VARIANTS: A GRAPH TRANSFORMATIONS BASED APPROACH

Khaled Khalfaoui[1], Allaoua Chaoui[2], Cherif Foudil[3] and Elhillali Kerkouche[4]

[1]Department of Computer Science, University of Jijel, Algeria
khalfaoui_kh@yahoo.fr
[2]Department of Computer Science and its applications, University of Constantine 2, Constantine, Algeria
a_chaoui2001@yahoo.com
[3]Department of Computer Science, University of Biskra, Algeria
foud_cherif@yahoo.fr
[4]Department of Computer Science, University of Jijel, Algeria
elhillalik@yahoo.fr

ABSTRACT

A Software Product Line is a set of software products that share a number of core properties but also differ in others. Differences and commonalities between products are typically described in terms of features. A software product line is usually modeled with a feature diagram, describing the set of features and specifying the constraints and relationships between these features. Each product is defined as a set of features. In this area of research, a key challenge is to ensure correctness and safety of these products. There is an increasing need for automatic tools that can support feature diagram analysis, particularly with a large number of features that modern software systems may have. In this paper, we propose using model transformations an automatic approach to validate products according to dependencies defined in the feature diagram. We first introduce the necessary meta-models. Then, we present the used graph grammars to perform automatically this task using the AToM3 tool. Finally, we show the feasibility of our proposal by means of running examples.

KEYWORDS

Software Product Lines, Feature Diagram, Variability Modelling, Structural Validation, Graph Transformations

1. INTRODUCTION

Software product line (SPL) engineering is an approach for developing families of software systems. The main advantage over traditional approaches is that all products can be developed and maintained together. This technique has found a broad adoption in several branches of software production. Feature models [1] are widely used in domain engineering to capture common and variant features among the different variants. A Feature Diagram is a hierarchically structured model that defines the features and their relationships. Each product is defined as a combination of features. One of the major problems is the validation of products from a structural viewpoint. Generally, this task becomes difficult with a large number of features. To remedy this

problem, development of automatic tools for verification proves necessary. In this research area, we are interested by model transformations approach [2].
Model transformations are a very useful in the evaluation, validation, manipulation and processing of diagrams. They are performed by executing graph grammars [3]. A graph grammar is composed of rules. Each one has a graph in their left and right hand sides (LHS and RHS). Rules are compared with an input graph called host graph. If a matching is found between the LHS of a rule and a subgraph in the host graph, then the rule can be applied and the matching subgraph of the host graph is replaced by the RHS of the rule. Furthermore, each rule may also have application conditions that must be satisfied, as well as actions to be performed when the rule is executed. A graph rewriting system iteratively applies rules of grammar in the host graph, until no rules are applicable.

In this paper, we propose an automatic framework based on this technique to check the validity of SPL products according to the dependencies defined in the feature diagram (FD). First, we verify parental relationships by exploring the FD tree in an up-bottom manner from the root to leaves. Then, we deal with the cross-Tree constraints. Analysis results will be edited in a text file.

The remainder of this paper is organized as follows: In section 2, we discuss some related works. Section 3 provides the background of our approach. We recall some basic notions about FD diagrams and give an overview of graph transformations technique. In Section 4, we present our proposal. We illustrate our framework through some examples in section 5. Finally, section6 concludes the paper and gives some perspectives of this work.

2. RELATED WORK

Research in the field of SPL is becoming increasingly important, particularly through its ability to increase software reuse. The success of this approach is conditioned by the correctness of final products. With a high degree of variability, automated analyses and verification are crucial. Over the past few years, a great variety of proposals had been proposed.

Mannion in [4] proposed the adoption of propositional formula for formally representing software product lines. The principal idea is that variability and commonality are translated into a propositional formula where the atoms represent features and the formula is valid if and only if a given configuration is admissible. This idea has been extended by creating a connection between feature models, grammars and propositional formula by Batory [5]. In [6], context-free grammars have been used to represent the semantics of cardinality-based feature models. This semantic interpretation of a feature model relates with an interpretation of the sentences recognized as valid by the context-free grammar. Sun et al. in [7] proposed a formalization of FMs using Z and the use of Alloy Analyzer for the automated support of the analyses of FMs. Wang et al. in [8] have proposed an approach to modeling and verifying feature diagrams using semantic Web Ontology Language (OWL). The authors have deployed OWL reasoning engines to check for the inconsistencies of feature configurations fully automatically. The specialists in the field asserts that three of the most promising proposals for the automated analysis of feature models are based on the mapping of feature models into Constraint Satisfaction Problem (CSP) solvers[9], propositional SAT is fiability problem(SAT) solvers [10] and Binary Decision Diagrams (BDD) [11]. The basic idea in CSP solvers is to find states where all constraints are satisfied. Somewhat similar to CSP solvers, SAT solvers attempt to decide whether a given propositional formula is satisfiable or not, that is, a set of logical values can be assigned to its variables in such a way that makes the formula true. Also, BDD is a data structure for representing the possible configuration space of a Boolean function, which can be useful for mapping a feature model configuration space. For more details see [12], where Benavides presented a comprehensive literature review on the most important techniques and tools.

Nowadays, graph transformations are widely used for modelling and analysis of complex systems in the area of software engineering [13]. In [14], the authors have proposed a tool that formally transforms dynamic behaviours of systems expressed using Unified Modelling Language (UML) Statechart and collaboration diagrams into their equivalent colored Petri nets models. Zambon et al. in [15] have proposed an approach for the verification of software written in imperative programming languages. The treatment was based on model checking of graph transition systems. They used an explicit representation of program states as graphs, and they specified the computational engine as graph transformations. In [16], it has proposed an approach to extract and integrate the parallel changes made to Object-Oriented formal specifications in a collaborative development environment. This approach allows combining the parallel changes made while addressing any merging conflicts at the same time. The authors in [17] have proposed an automatic approach to check UML models using the graph transformations approach. The idea was to map Class Diagrams, StateCharts and Communication Diagrams into a single Maude specification. Properties verification is performed using the Linear Temporal Logic (LTL) Model Checker.

In previous work [18], we have proposed an automatic approach for behavioural analysis of SPL products. The current paper concerns the structural validation.

3. BACKGROUND

3.1. Feature Modelling

Research on feature modeling has received much attention in software product line engineering community. Feature-Oriented Domain Analysis (FODA) [1] is the most the most popular. The success of this approach resides in the introduction of feature models, which contain a graphical tree-like notation that shows the hierarchical organization of features. In the tree, nodes represent features; edges describe feature relations. A single root node represents the domain concept being modeled. Figure.1 depicts a simplified feature model of a mobile phone SPL.

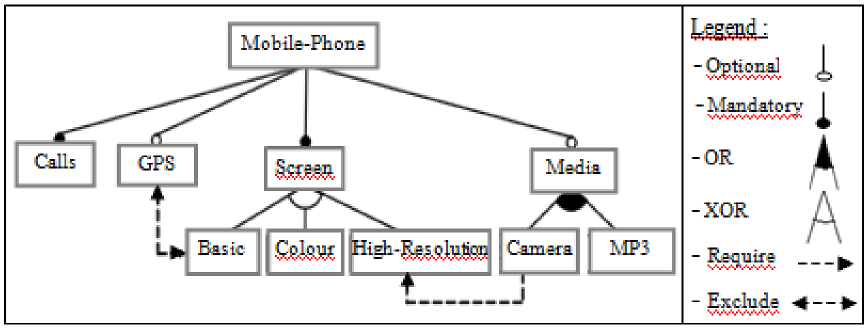


Figure 1. Feature diagram of a mobile phone product line

Current feature modeling notations may be divided into three main groups: Basic feature models, Cardinality-based feature models and Extended feature models. In this work, we are only interested on basic feature models. Relationships between a parent feature and its child features (or subfeatures) are:

- Mandatory: If the father feature is selected, the child feature must be selected.
- Optional: If the father feature is selected, the child feature may be selected but not necessarily.

- XOR: If the father feature is selected, exactly one feature of the children features must be selected.
- OR: If the father feature is selected, at last one feature of the OR-child features must be selected.

In addition, cross-tree constraints are allowed. The most common are:

- Require: The selection of source feature implies the selection of the destination.
- Exclude: both features cannot be part of the same product.

The success of software product line approach is conditioned by the correctness of final products. Modeling errors will inevitably affect the following steps. At this level, we must ensure that the products are valid of a structural point of view. From the Mobile phone feature model (Fig.1), consider the following configurations:

- *P_1: Mobile-Phone, Calls, Screen, Basic, Media, MP3*
- *P_2: Mobile-Phone, Calls, GPS, Screen, Basic, Media, MP3*

We remark that P_1 is correct; however P_2 is invalid.

3.2. Model Transformations

Raising the abstraction level from textual programming languages to visual modeling languages, model transformation techniques have become more focused recently. They are successfully applied in several domains. The translation is performed by executing graph grammars which are a generalization of Chomsky grammars for graphs [2]. A graph transformation rule (Figure.2) is a special pair of pattern graphs called left hand side (LHS) and right hand side (RHS). They are defined such that an instance defined by the LHS is substituted with the instance defined in the RHS when applying such rule. Rules are local in a sense that they handle only a small amount of model elements, and therefore the designer does not need to concentrate on the entire transformation problem.

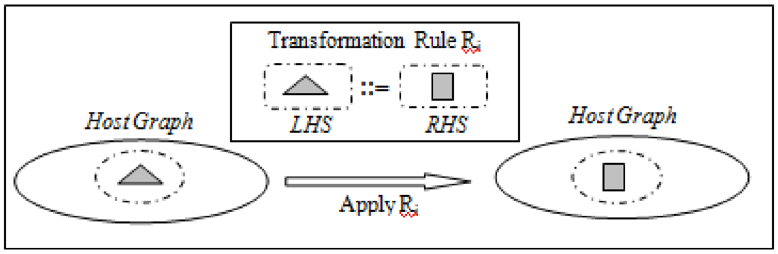


Figure 2. Graph transformation rule

In the rewriting process, rules are evaluated against an input graph, called the host graph. If a matching is found between the LHS of a rule and a subgraph of the host graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Generally, rules are ordered according to a priority assigned by the user and are checked from the higher priority to the lower priority. After a rule matching and subsequent application, the graph rewriting system starts again the search. The graph grammar execution ends when no more matching rules are found.

In the field of graph transformation, the meta-modeling technique is widely used to describe the different kinds of formalisms needed in the specification and design of systems. To define a meta-model, we have to provide two syntaxes. On one hand, the abstract formal syntax to denote the formalism's entities, their attributes, their relationships and the constraints. On the other hand, the concrete graphical syntax to define graphical appearance of these entities and relationships. The advantage of this technique is that the generated tool accepts only syntactically correct models according to the formalism definition.

AToM^3 [18] is a visual tool for meta-modeling and model transformations. Its meta-layer allows a high-level description of models using the Entity-Relationship (ER) formalism extended with the ability to specify the graphical appearance. Once the meta-model of a given formalism is defined, AToM^3 generates automatically an interactive environment to visually manipulate (create and edit) models. In the LHS of rules, the attributes of the nodes must be provided with values which will be compared with the nodes attributes of the host graph during the matching process. These attributes can be set to <ANY> or have specific values. In order to specify the mapping between LHS and RHS, nodes in both LHS and RHS are identified by means of labels (numbers). If a node label appears in the LHS of a rule, but not in the RHS, then the node is deleted when the rule is applied. Conversely, if a node label appears in the RHS but not in the LHS, then the node is created when the rule is applied. If a node is created or modified by a rule, we must specify in the RHS the appropriate Python code to calculate its attributes' values. In addition, AToM^3 allows the use of global attributes accessible in all of the graph grammar rules.

4. A GRAPH TRANSFORMATION APPROACH FOR STRUCTURAL VALIDATION OF SPL PRODUCTS

In this section we present our proposal. To facilitate the processing, we prefer at first translating the FD diagram into a decorated tree noted D-Tree. The purpose is to obtain a model easier to explore. Then, the validation of a given product will be performed by dealing this D-Tree model. The verification result is edited in a text file. So, as seen in figure Figure.3, we have two graph transformations to realize.

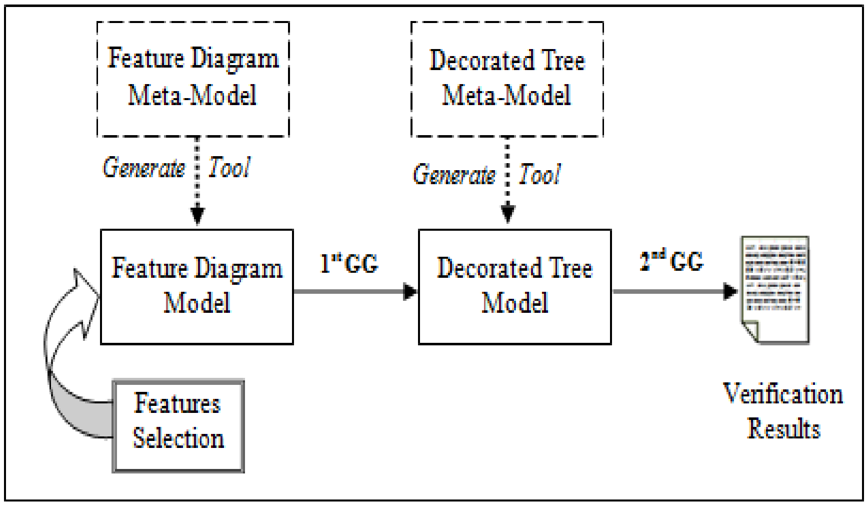


Figure 3. The general outline of the proposed approach

In the following, we first present the proposed meta-models for FD and D-Tree formalisms. Then, we introduce the used graph grammars.

5.1 Meta-Modeling

<u>FD meta-model:</u> It is composed of:

- Feature Entity: Each feature has three attributes:
    - Its identifier *Name*.
    - A boolean attribute called *isRoot* used to identify the root feature.
    - A boolean attribute called *isSelected* used to specify selected features.
- OR Relationship, XOR Relationship, Mandatory Relationship, Optional Relationship, Include Relationship and Exclude Relationship.

We note that all of these relationships don't have attributes, but differ in their graphic appearance. They are adjusted according to their appropriate notations.

<u>D-Tree meta-model:</u> It consists on:

- Node Entity: It has four attributes:
    - *Name:* to identify the node.
    - The same, two Booleans: *isRoot* and *isSelected.*
    - *Count-SelectedChilds*: used to specify the number of its selected children.
- GenericLink Relationship: This association represents links between nodes in the D-Tree model. Graphically, they have the same appearance. To differentiate them, we added an attribute called *RelationType*.

5.2 Defining the Graph Grammars

<u>1st GG: FD to D-Tree:</u>

The purpose of this grammar is to translate the FD diagram into an equivalent D-Tree model. In addition, it calculates the number of children for each feature. To perform this treatment, we propose a graph grammar with eight rules (Figure.4).

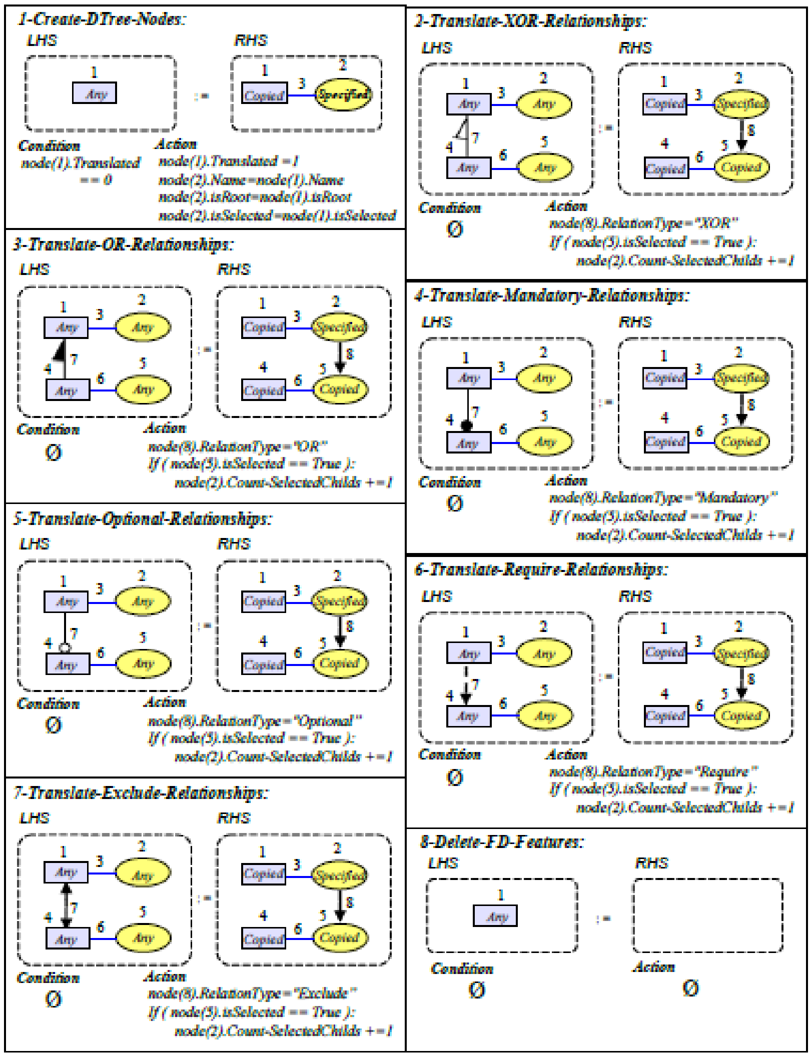

1-Create-DTree-Nodes:
LHS
RHS
Any
Copied
Specified
Condition
node(1).Translated == 0
Action
node(1).Translated = 1
node(2).Name=node(1).Name
node(2).isRoot=node(1).isRoot
node(2).isSelected=node(1).isSelected
2-Translate-XOR-Relationships:
node(8).RelationType="XOR"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
3-Translate-OR-Relationships:
node(8).RelationType="OR"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
4-Translate-Mandatory-Relationships:
node(8).RelationType="Mandatory"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
5-Translate-Optional-Relationships:
node(8).RelationType="Optional"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
6-Translate-Require-Relationships:
node(8).RelationType="Require"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
7-Translate-Exclude-Relationships:
node(8).RelationType="Exclude"
If (node(5).isSelected == True):
node(2).Count-SelectedChilds += 1
8-Delete-FD-Features:
Condition
Ø
Action
Ø

Figure 4. FD to D-Tree graph grammar

Treatment begins with the creation of the D-Tree nodes. Each time the rewriting system locates an FD feature and associates it to a new D-Tree node. The attributes *Name*, *isRoot* and *isSelected* are copied with the same values. To do this, we use a temporary attribute called *Translated* to indicate whether each FD feature has been previously treated or not. Then, we move on the creation of the D-Tree links. At this level, each FD relationship is transformed into a D-Tree GenericLink. The attribute *RelationType* is set according to the FD relationship type. At the same time, for each node, we count the number of the selected children. It will be used in the next step. Finally, we clean the created D-Tree model of the FD features.

The execution of rules N°2, N°3, N°4, N°5, N°6 and N°7 is not subject to any condition. This is because in the application of each one, there is always deleting of the FD relationship, and therefore this treatment does not reproduce. This allowed us to avoid the use of other temporary attributes and get rid FD relationships at same time.

2nd GG: Validating-SPL-Products

This graph grammar model acts on the obtained D-Tree model. Its purpose is to analyze the product in question by checking all dependences. The result of this verification is edited in a text file. We propose to perform this treatment in two steps:

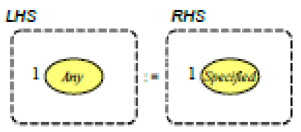
Step1: In the first step, we deal only parental relationships. For each node, if selected, we have to check validity of its children according to their paternal relationship. If this latter is not satisfied, the error will be edited in the text file. Once all children treated, the selected ones will be treated as parents and we start again the same treatment. So, we have to explore the D-Tree model from top to bottom, starting with the root up to the leaves. To do this, we use the following auxiliary attributes:

- *Current-Parent*: is used to identify the node currently being treated as parent.
- *isTreated-AsParent*: is used to indicate whether this node has been treated as parent or not.
- *isTreated-AsChild*: is used to indicate whether this child has been visited during treatment of its parent or not.
- *ToTreat-AsParent*: is used to indicate whether this node should be treated as parent or not.

The number of the selected children is used only to validate Or and XOR relationships, but once. To do this, we use an attribute called *NumChilds-Checked*.

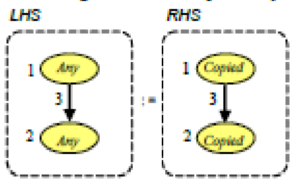
Step2: Now, we treat the cross-tree constraints. To check them one by one, we add to the GenericLink associations an auxiliary attribute called *isChecked*. It is used to indicate whether a Require or Exclude link has already been verified or not. Similarly, in the case of anomaly, error will be edited in the text file. To carry out this process, we propose eleven rules (Figure.5).

1- Initialization:



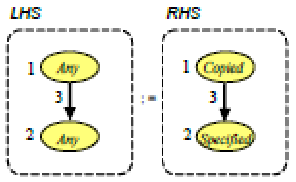
```
Condition
  node(1).isRoot == True and
  node(1).ToTreat-AsParent == 0 and
  node(1). isTreated-AsParent == 0 and

Action
  node(1).ToTreat-AsParent =1
```

2- Checking OR-Relationship-Validity:



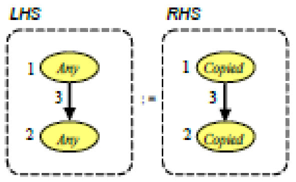
```
Condition
node(1).Current-Parent = 1 and
node(3).RelationType = = "OR"
NumChilds-Checked == 0

Action
If (node(1).Count-SelectedChilds == 0):
      TextFile.wrinting("Error: For "+ node(1).Name +
      "node , at least one child must be selected")
NumChilds-Checked =1
```

3- OR-Relationship-Processing:



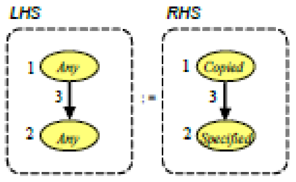
```
Condition
node(1).Current-Parent = 1 and
node(2).isTreated-AsChild == 0 and
node(3).RelationType = = "OR"

Action
node(2).isTreated-AsChild = 1
If (node(2).isSelected == True ):
      node(2).ToTreat-AsParent =1
```

4- Checking XOR-Relationship-Validity:



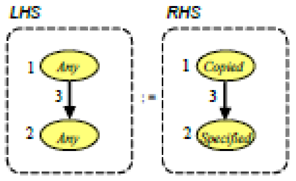
```
Condition
node(1).Current-Parent = 1 and
node(3).RelationType = = "XOR"
NumChilds-Checked == 0

Action
If (node(1).Count-SelectedChilds !=1):
      TextFile.wrinting("Error: For "+node(1).Name +
      "node , exactly one child must be selected")
NumChilds-Checked = 1
```

5- XOR-Relationship-Processing:

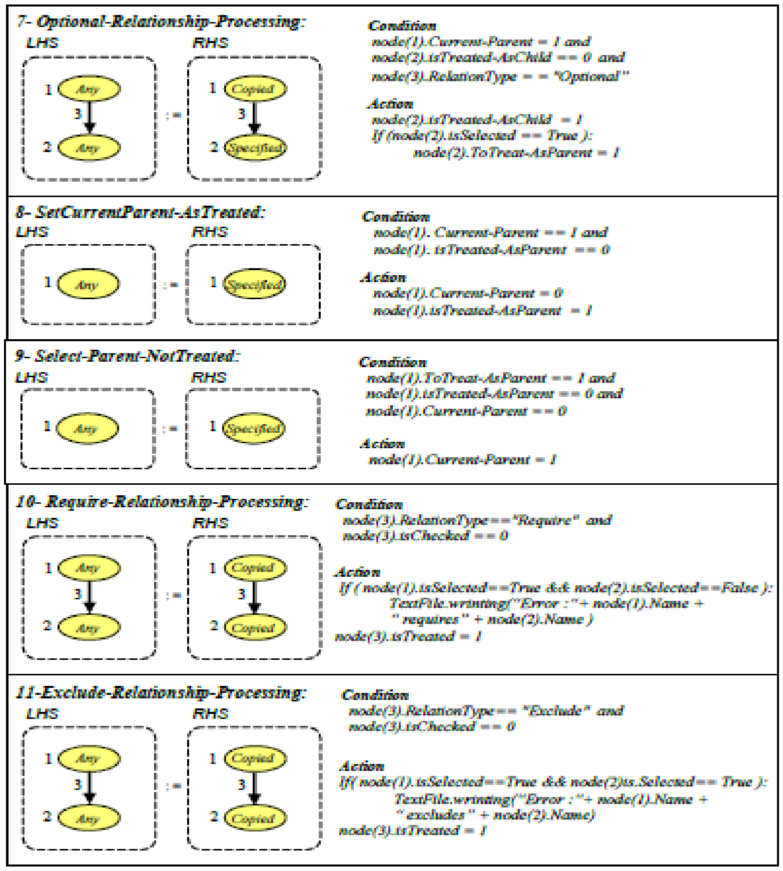


```
Condition
node(1).Current-Parent = 1 and
node(2).isTreated-AsChild == 0 and
node(3).RelationType = = "XOR"

Action
node(2).isTreated-AsChild = 1
If (node(2).isSelected == True ):
      node(2).ToTreat-AsParent =1
```

6- Mandatory-Relationship-Processing:



```
Condition
node(1).Current-Parent = 1 and
node(2).isTreated-AsChild == 0 and
node(3).RelationType = = "Mandatory"

Action
If (node(2).isSelected == False):
      TextFile.wrinting(Error: "+ node(2).Name +
      " is mandatory. It must be selected.")
node(2).isTreated-AsChild = 1
node(2).ToTreat-AsParent = 1
```



7- Optional-Relationship-Processing:

```
Condition
  node(1).Current-Parent = 1 and
  node(2).isTreated-AsChild == 0 and
  node(3).RelationType = = "Optional"

Action
  node(2).isTreated-AsChild = 1
  If (node(2).isSelected == True ):
          node(2).ToTreat-AsParent = 1
```

8- SetCurrentParent-AsTreated:

```
Condition
  node(1). Current-Parent == 1 and
  node(1). isTreated-AsParent == 0

Action
  node(1).Current-Parent = 0
  node(1).isTreated-AsParent = 1
```

9- Select-Parent-NotTreated:

```
Condition
  node(1).ToTreat-AsParent == 1 and
  node(1).isTreated-AsParent == 0 and
  node(1).Current-Parent == 0

Action
  node(1).Current-Parent = 1
```

10- Require-Relationship-Processing:

```
Condition
  node(3).RelationType=="Require" and
  node(3).isChecked == 0

Action
If ( node(1).isSelected==True && node(2).isSelected==False ):
        TextFile.wrinting("Error :"+ node(1).Name +
        " requires" + node(2).Name )
node(3).isTreated = 1
```

11-Exclude-Relationship-Processing:

```
Condition
  node(3).RelationType== "Exclude" and
  node(3).isChecked == 0

Action
If( node(1).isSelected==True && node(2)is.Selected== True ):
        TextFile.wrinting("Error :"+ node(1).Name +
        " excludes" + node(2).Name)
node(3).isTreated = 1
```

The execution of the first nine rules realizes $Step_1$, while the application of rules N°10 and N°11 performs $Step_2$.

6. Illustrative Examples

To illustrate our framework, let us consider the Mobile Phone example presented previously in Section 3.

First, we have to create the FD model (Figure.6).

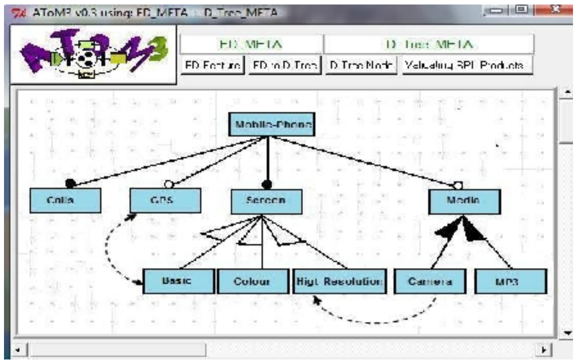


Figure 6. FD diagram

Then, we select the features of the product that we going validate. Consider the product:

- *P_1: Mobile-Phone, GPS, Screen, Basic, Media.*

Using this environment, to specify this product we have to set as true the attribute *isSelected* of these features.

By executing the first graph grammar, we obtain the corresponding D-Tree model (Figure.7).

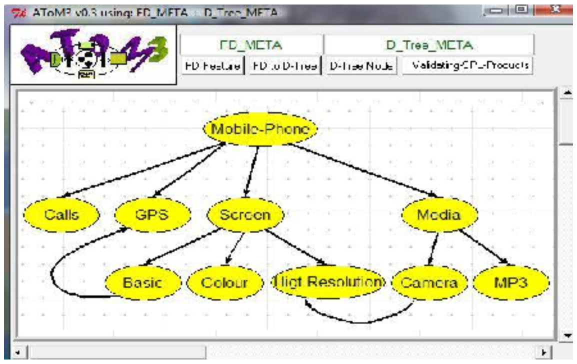


Figure 7. The generated D-Tree model

To perform automatically the validation of this product, we have to execute the second graph grammar on the obtained D-Tree model. The text file containing the results of this analysis is presented in Figure.8.

```
- Error: Calls is mandatory feature. It must be selected.
- Error: GPS excludes Basic.
- Error: For Media node, at least one child must be selected.
```

Figure 8. P_1 validation results

It shows that there are three errors:

- The first concerns the node Calls. It is a mandatory feature, but it is not selected.
- The second is the fact that GPS and Basic features are linked by an Exclude relationship, whereas they are both selected.
- The last error regards the node Camera. This feature is selected, but none of its children is selected. There must be at least one.

Consider now the product:

- *P_2: Mobile-Phone, Calls, Screen, Basic, Colour, Media, Camera.*

By following same steps, we obtained the text file presented in Figure.9. It shows that P_2 is invalid since:

- The features Basic and Colour are exclusive to each other, but both are selected.
- The feature Camera necessities the presence of High-Resolution.

```
- Error: For Screen node, exactly one child must be selected.
- Error: Camera requires High-Resolution.
```

Figure 9. P_2 validation results

Finally, for the product:

- *P_3: Mobile-Phone, Calls, GPS, Screen, High-Resolution, Media, Camera, MP3.*
-

The generated text file is empty. There are no errors, therefore this product is valid.

7. CONCLUSION

Software product line engineering is about producing a set of related products that share more commonalities than variabilities. Feature models are widely used in domain engineering to capture common and variant features among the different variants. Each legal product is defined as a combination of features that respects all dependencies defined in the FD model. With a large number of features, the structural validation of products is extremely difficult. In fact, we have to verify satisfaction of all relationships and constraints defined in the FD model.

To remedy this problem, we have proposed a novel approach based on graph transformations with two steps. In the first, we have treated the parental relationships by exploring the FD tree in an up- bottom manner from the root to leaves. For each selected feature, the relationship with its children is checked according to the selected ones. The second step is dedicated to verify the cross-tree dependencies. Constraints that are not satisfied are detected. To do this, we have proposed two graph grammars. The first is used to translate the FD diagram into an equivalent decorated tree in which all relationships between nodes are of the same type. The second performs the verification by dealing the D-Tree model. The analysis results are generated in a text file. The choice of the graph transformations technique is motivated by the fact that:

- It constitutes the most appropriate solution which allowed us the exploration of the FD diagram easily.
- As this verification is based on local computations, it was found that graph grammar rules are the most suitable solution.
- It is implemented directly as a fully automatic tool.
- It is extensible.

In our future work, we plan to develop an integrated environment to generate all valid products according to the dependencies specified in the feature diagram.

REFERENCES

[1] K. Kang, S. Cohen, J. Hess, W. Novak and S. Peterson, (1990) "Feature-oriented domain analysis (FODA) feasibility study", Technical Report, CMU/SEI- 90-TR-21.

[2] M. Andries, G. Engels, A. Habel, B. Hoffmann, H. J. Kreowski, S. Kuske, D. Pump, A. Schürr and G. Taentzer, (1999) "Graph transformation for specification and programming", Science of Computer Programming, Vol. 34, No. 1, pp 1-54.

[3] G. Rozenberg, (1999) "Handbook of graph grammars and computing by graph transformation", World Scientific, Singapore, Vol. 1.

[4] M. Mannion, (2002) "Using first-order logic for product line model validation", SPLC 2, Chastek, G.J. (Ed.), Springer-Verlag, London, pp. 176-187.

[5] D. Batory, (2005) "Feature models, grammars, and propositional formulas", Lecture notes in Computer Science, Vol. 3714, pp. 7-20.

[6] K. Czarnecki,, S. Helsen, U. Eisenecker, (2004) "Staged configuration using feature models", In SPLC 2004, Heidelberg, LNCS, Vol. 3154, pp. 266-283.

[7] J. Sun, H. Zhang, Y.F. Li and H. Wang, (2005) "Formal semantics and verification for feature modeling", In Proceedings of the ICECSS05, pp. 303-312.

[8] H. Wang, Y.Li, J. Sun, H. Zhang and J. Pan, (2007) "Verifying feature models using OWL", Web Semantics: Science Services and Agents on the World Wide Web, Vol. 5, No. 2, pp. 117-129.

[9] D. Benavides, P. Trinidad and A. Ruiz-Cortes, (2005) "Automated reasoning on feature models", in CAiSE 2005, LNCS, Springer, Vol. 3520, pp. 491-503.

[10] E. Bagheri, T.D. Noia, D. Gasevic and A. Ragone, (2012) "Formalizing interactive staged feature model configuration", Journal of Software: Evolution and Process, Vol. 24, No. 4, pp. 375-400.

[11] M. Mendonca, A.Wasowski, K. Czarnecki, and D. Cowan, (2008) "Efficient compilation techniques for large scale feature models", in Proceedings of GPCE '08, USA, ACM Press, pp.13-22.

[12] D. Benavides, S. Segura, A. Ruiz-Cortés, (2010) "Automated Analysis of Feature Models 20 Years Later: A Literature Review", Journal of Information Systems, Vol. 35, No. 6, pp. 615-636.

[13] H. Ehrig, G. Engels, H. J. Kreowski, G. Rozenberg, (2012) "Graph Transformation", Sixth International Conference on Graph Transformation, ICGT 2012, LNCS, Springer, Vol. 7562.

[14] E. Kerkouche, A. Chaoui, E. B. Bourennane and O. Labbani, (2010) "On the use of graph transformation in the modeling and verification of dynamic behavior in UML models", JSW, Vol. 5, No. 11, pp. 1279-1291.

[15] E. Zambon and A. Rensink, (2011) "Using Graph Transformations and Graph Abstractions for Software Verification", ICGT-DS, Electronic Communications of the EASST, Vol. 38.

[16] F. Taibi, (2012) "Automatic Extraction and Integration of Changes in Shared Software Specifications", International Journal of Software Engineering and Its Applications, Vol. 6, No. 1, pp. 29-45.

[17] W. Chama, R. ElMansouri and A. Chaoui, (2012) "Model Checking and Code Generation for UML Diagrams using Graph Transformation", International Journal of Software Engineering & Applications (IJSEA), Vol. 3, No. 6, pp. 39-55.

[18] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, (2012) "Formal Specification of Software Product Lines: A Graph Transformation Based Approach", JSW, Vol. 7, No. 11, pp. 2518-2532.

[19] J. De Lara and H. Vangheluwe, (2002) "AToM3: a tool for multi-formalism modelling and meta-modelling", Lecture Notes in Computer Science, Vol. 2306, pp.174-188.