Article

Automated inconsistency detection in feature models: A generative programming based approach

Muhammad Javed[1], **Muhammad Naeem**[2], **Aarif Iqbal Umar**[1], **Faisal Bahadur**[1]
[1]Department of Information Technology, Hazara University, Mansehra, Pakistan
[2]Department of Information Technology, Abbottabad University of Science & Technology, Havelian, Pakistan
E-mail: mjavedgohar@hotmail.com, naeem@hu.edu.pk, aarifiqbalumar@yahoo.com, msosfaisal@yahoo.com

Received 5 March 2016; *Accepted 10 April 2015*; *Published online 1 June 2016*

Abstract
The quality of feature model represents the quality of end products because it is used to develop products. Hence, quality evaluation of feature model is the most important task. The quality detection mechanism should be efficient enough to evaluate the quality of a given feature model within limited time. So, there is a need of automated quality evaluation system. Generative Programming (GP) is the most effective way to automate the quality detection system for feature models. This effort is to present an efficient way to automate the quality detection system by using one of the GP based technique (GenVoca Layered Architecture) for inconsistencies in feature model. We implemented this quality detection technique in C++. We applied this technique on the feature models contain errors.

Keywords quality of feature models; maturity model; Generative Programming (GP); inconsistencies; GenVoca Layered Architecture.

Selforganizology
ISSN 2410-0080
URL: http://www.iaees.org/publications/journals/selforganizology/online-version.asp
RSS: http://www.iaees.org/publications/journals/selforganizology/rss.xml
E-mail: selforganizology@iaees.org
Editor-in-Chief: WenJun Zhang
Publisher: International Academy of Ecology and Environmental Sciences

1 Introduction

Producing things in large amount require standardized processes, especially for the similar products. Companies are organizing their production in large amount of production (Benavides et al., 2010). To reuse existing systems in a systematic way, service-oriented systems resemble supply chain where products manufactured from supplied parts. Same case is for complex service-oriented systems, that needs third party services (Thomas, 2008). For example, car producer offer variation on a model with variable engines, gearboxes, audio and entertainment systems. Example of software services is online travel agency that may use third-party services for hotel booking, invoicing and for payment option (Naeem, 2012). Similarly, increasing number of software systems with almost similar requirements guide us to Software Product Line (SPL) (Böckle and Linden, 2005). SPL Engineering helps in the development within application domain by considering their commonalities and variability. In SPL approach, products are being created by reusability (Clements and Linda, 2002). SPL incorporating the property of similarities and variability in the family of

software is a new technique in the development of software. This helps in the development of high quality software in a short period of time with low budget. Progress has been improved in the development by adopting SPL (Mendonça, 2009). Features represent the aspects of these software (Kang et al., 1990). In mostly software systems user can select or deselect functional and nonfunctional properties and these options available to users are known as features (Batory et al., 2006).To get a valid combination of these features we use feature model that depicts the relationships of these features and constraints on them (Batory, 2005).

The use of high quality process ensures the good quality resulting products. Hence, it is very important to investigate the quality of the selected model before putting it into practice. In other words, one can say that the quality of a feature model has prime importance because it contributes towards the development of high quality products. There are number of properties affecting the quality of a feature model. One of the agreed deficiencies is the presence of inconsistencies in the feature model.

This paper is to automate quality detection process by one of the Generative Programming (GP) techniques.GP has been adopted in domain analysis since the 80's (reference). In GP, system can be produced from requirements written in domain-specific languages. GP is to develop new systems easily on the basis of reusable components (Lung et al., 2010).GP emphases on designing and implementation of reusable software instead of developing software separately for each problem. Hence, the target of generative analysis and design are families of systems (Czarnecki and Eisenecker, 2000). Main objectives of GP are: minimizing the gap between concept and implementation, attain high level of reusability, easy to manage different components and increasing efficiency (Czarnecki et al., 2000).

The rest of the paper is arranged as follows: Sections 2 and 3provide the background information and the related discussions, respectively. Section 4 contains the proposed GenVoca layered architecture based inconsistency detection technique and in Section 5 defines the evaluation method of proposed technique.

2 Background

Feature models were introduced by Kang in the form of a technical report on FODA in 1990. A feature is prominent characteristic of a product (Kang et al., 1990). Feature model is a hierarchical model that captures the commonality and variability of SPL. The set of permissible selection of features from a feature model is called an instance (Rosso, 2006). The selection of a feature from a feature model is based on the relevance with its parent feature.

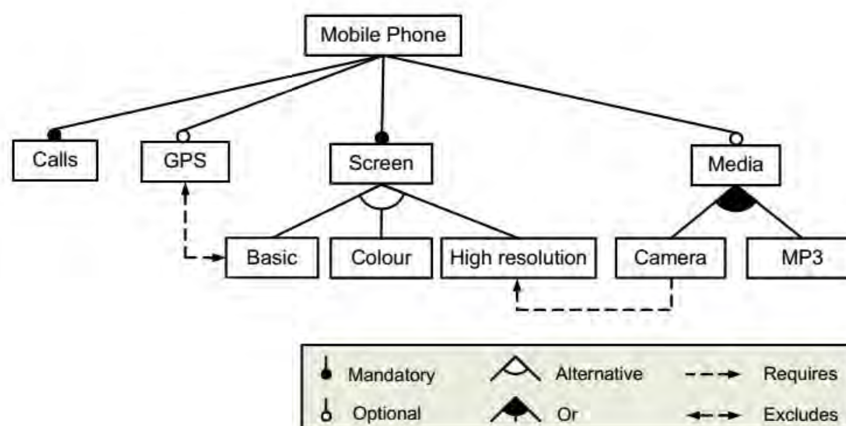

Fig. 1 A feature model of a mobile phone (Benavides et al., 2010).

A feature model offers the following list of features:

Mandatory feature: If a feature is chosen then its mandatory feature must be selected in that instance (Benavides et al., 2010). It is represented by a filled circle at the end of edge. For example, *Call* is a mandatory feature of *Mobile Phone* in Fig. 1.

Optional feature: If a feature is selected in an instance then its optional sub-features can be selected or rejected depending on the preferences (Benavides et al., 2010). It is represented by empty circle at the end of edge. For example, *GPS* is an optional feature of *Mobile Phone* shown in Fig. 1.

Alternative-group: A group of features having an alternative relevance with their parent means that exactly one feature from this group must be selected if their parent is selected in an instance. It is represented by unfilled arc (Benavides et al., 2010). For example, features occurring under *Screen* make an Alternative-group in Fig. 1.

Or-group: For a group of features having an OR relevance with their parent means that at least one feature from this group must be selected, if their parent is selected in an instance (Rosso, 2006). An Or-group is shown by a filled arc. For example, features occurring under *Media* are making an Or-group in Fig. 1.

Apart from the parent child relationship, a feature diagram may have cross-tree constraints that are discussed below

Requires constraint: If a source of requires constraint is selected that its target must also be chosen in that instance. This is represented by the dashed arrow that starts from the source and heads towards the target feature. For example, requires constraint is shown between *Camera* and *High resolution* features in Fig. 1.

Excludes Constraint: The source and target features of excludes constraint cannot be selected in an instance. This is represented by double headed dashed arrow, as shown between *Basic* and *GPS* features in Fig. 1.

2.1 Inconsistency in feature model

Inconsistency arises due to the conflicting information in a feature model. It is impossible to obtain any valid instance from inconsistent feature models. So, inconsistencies are characterized as critical error (Maßen and Horst, 2004). Following are the inconsistency based errors.

Void feature models: A void feature model defines no instance, i.e., no feature can be selected. This means that each feature is dead including the root. Thus we say that a void feature model is the one whose root is a dead feature (Trinidad et al., 2008). In Fig. 2, some of the void feature models are presented.

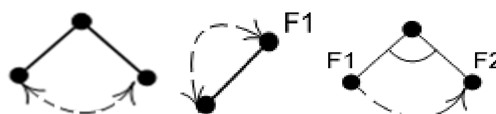

Fig. 2 Examples of void feature model (Maßen and Horst, 2004).

Invalid Product: Invalid product means that invalid instance of a feature model (Benavides et al., 2010). Invalid instance misses at least one required feature, e.g., mandatory feature of a feature model (Segura et al., 2010). In Fig. 3, a mandatory feature *E* cannot be chosen due to the presence of implies constraint on multiple features that depicted under one alternative set.

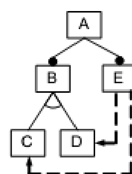

Fig. 3 Feature model with invalid products (Segura et al., 2010).

2.2 GenVoca Architecture an approach for Generative Programming

In GenVoca Model features are represented in layers (Czarnecki and Eisenecker, 2000). For instance the following expression from Fig. 1 describe that *Mobile Phone* contains *Calls* and *Screen* functionality with *High resolution*

Mobile Phone[calls[screen[high resolution]]]

The following figure represents the layered architecture of the above expression where each layer represents a class with attributes and methods (Czarnecki and Eisenecker, 2000).

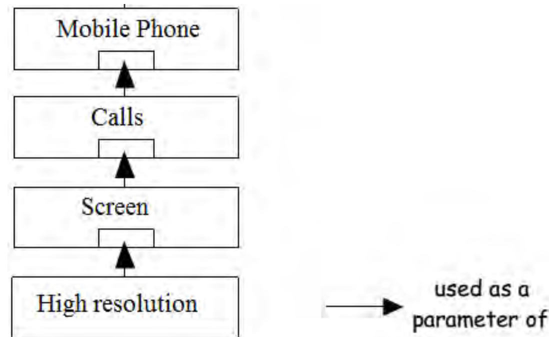

Fig. 4 GenVoca Layers (Czarnecki and Eisenecker, 2000).

In this architecture each lower layer passes some parameters to the layer above and the above layer has interface to collect the parameters. In GenVoca each interface is known as realm. A realm is collection of classes (Czarnecki and Eisenecker, 2000). Class of GenVoca domain model can be represented as the following:

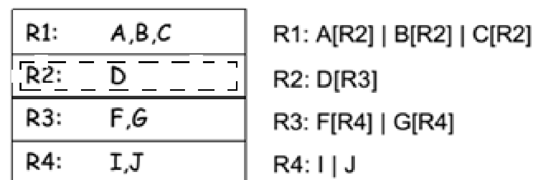

Fig. 5 Example of a stacking model in graphical notation and the corresponding grammar (Czarnecki and Eisenecker, 2000).

In Fig. 5 each box represents a layer and we can get an instance by selecting alternative from these layers. For example B[D[F[J]]] is valid instance selected from the Fig. 3. The dashed inner box represents the *optional* generic layer. GenVoca layers can be implemented in C++ as class templates containing member classes. Following figure represents the idea of implementing GenVoca layers in C++ (Czarnecki and Eisenecker, 2000).

```
template <class LowerLayer>
class LayerA
{   public:
        class ClassA
        {   public:
                LowerLayed::ClassA lower;

                // refine operationA()
                void operationA()
                {   ... // LayerA-specific work
                    lower.operationA();
                    ... // LayerA-specific work
                };

                //forward operationB()
                void operationB()
                {   lower.operationB(); };
        };

        class ClassB
        { ... };
};
```

Fig. 6 Implementation of GenVoca layer in C++ (Czarnecki and Eisenecker, 2000).

Layers can be implemented as structure instead of classes when all members are public (Czarnecki and Eisenecker, 2000). Structure based upward propagation in GenVoca model is as the following:

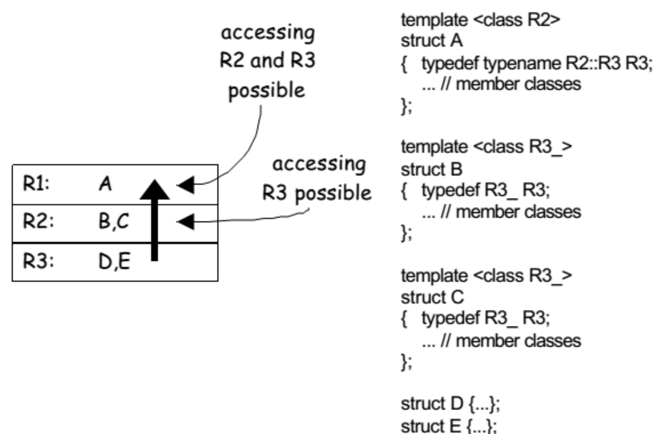

Fig. 7 Upward type propagation in a GenVoca model (Czarnecki and Eisenecker, 2000).

In this model each layer requires parameter from the layer below it so, the layer has to pass parameters to the upper layer even though lower layer do not need any parameters for itself and this results in a problem when changing layers. So, to address this problem a standard “envelope” has to pass to all layers and this envelope is known as *configurator class* or *config class.* This config class is passed to all layers of GenVoca domain model. All communication between layers can be done through this config class. This config class contains the configuration information of all features (Czarnecki and Eisenecker, 2000).

3 Related Work

3.1 Components and Generative Programming

In Czarnecki and Eisenecker (1999), authors presented a GenVoca model based automated approach for selection and configuration of features. They used the software product line based example of a car feature model. Requirements mapped into configuration after the selection of correct components. The main thing in this automation process is configuration knowledge and this converts the problem space to the solution space. The solution space contains implementation components with all possible components to maximize combinability and reusability. The problem space contains domain concepts and the required features for implementation. The configuration knowledge contains features combinations that are not allowed, default settings, default dependencies and construction rules. GenVoca model implemented with the steps of identification of main functionality, categorizing the components, identification of dependencies between components and representing the categories in a layered architecture. After defining grammar we implemented the selected example components implemented in C++ by bottom-up approach.

3.2 Experience of building an architecture-based generator using GenVocafor distributed systems

In this paper implemented generative programming by using GenVoca model approach by taking a distributed computing based case study. This helped in automation and reusability (Lung et al., 2010). This case study contains different distributed architectures i.e. Single Thread (ST), Half-Sync/Half-Async (HS/HA), and Leader/Followers (LFs). After the implementation using GenVoca Model the generator can instantiate the required system from the pattern. The development of generator is based on existing client/server (C/S) and Peer-to-Peer (P2P) systems. The generator developed by reserves engineering and forward engineering to build reusable components, GenVoca layered structure and corresponding implementation. The tasks carried out in re-engineering were reverse engineering of existing system and forward engineering (Lung et al., 2010).

The reverse engineering used to understand the problem space and the solution space. In this step both computing models (C/S and P2P) with three pattern (ST, HA/HA and LFs) studied and main features identified. In forward engineering, feature model of computing models with their alternatives developed. To construct GenVoca layering structure, the identified commonalities and variablities were implemented. In forward engineering step GenVoca Model implemented by considering the basic steps components identification, defining the layers by adopting bottom-up approach and implementation of identified components in C++ (Lung et al., 2010).

In above discussed approaches authors implemented feature models in GP. These approaches contains neither cross tree constraints nor error detection mechanism. In our approach we are going to implement the inconsistencies based errors detection mechanism which was presented in previous paper (Javed et al., 2015). Earlier to this we defined levels for the quality of feature model based on errors in (Javed et al., 2014).

4 Generative Programming for Inconsistencies

In this section, we are presenting generator for inconsistencies detection from our selected examples. The generator is based on the errors defined in section 2.1, i.e., void feature model and invalid instance.

4.1 GenVoca Model for void feature model

The following figure depicts the void feature model because except root no other feature is selectable due to the cross tree constraint (Trinidad et al., 2008).

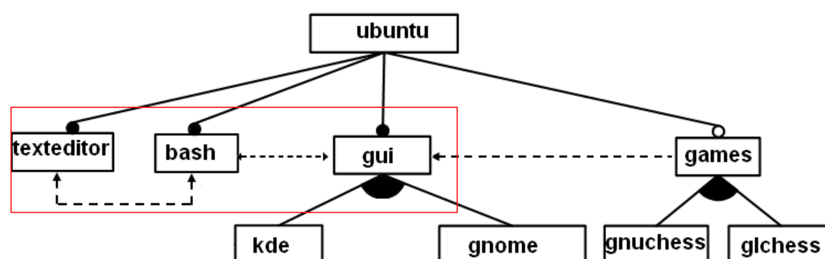

Fig. 8 example of void feature model (Felfernig et al., 2013).

In Fig. 8, features *texteditor, bash* and *gui* could not be chosen due to the presence of exclusion constraints between them, whereas the feature *games* will never get selected because of implies constraint with *gui*. On the other hand *vi, gedit, kde, gnome, gnuchess* and *glchess* are not selected because the selection of their respective parent features is not made. Only one feature, i.e., *ubuntu* can only be selected in the instance, this shows that this model is void feature model.

4.1.1 Identification and categorizing functionalities in feature model

To implement GenVoca model we have to identify the main functionalities from feature model that will be used to find dependencies, defining layers and for implementation. The followings are the main categories in our example depicted in Fig. 8.

Table 1 Functionalities of feature model in Fig. 8

Documentation	**File Handling**	**User Interface**	**Entertainment**
Texteditor	Bash	Kde	Gnuchess
		Gnome	Glchess

In above categories, headings represent the main functionalities of the product while the features under each heading are components to perform the required functionality, but instead of calling them components we will call them features to clear the concept. The first functionality is "Documentation" which will be performed by the feature *texteditor*. For "File Handling" functionality *Bash* feature will be used. For "User Interface" one of the features from *Kde* and *gnome* is required. Similarly for "Entertainment" we have feature, *Gnuchess* and *Glchess*.

4.1.2 Functional dependencies between features

In-order to perform the functionality some features depend on other. Identification of this dependency will help to model the layers. Because independent features will be on lower layer and dependent features will be on upper layer. Following are the dependencies between features.

texteditor depends on *gui*
games depends on *gui*
games depends on *bash*
bash depends on *gui*

After finding the dependencies we have to represent the functionalities in layers. This is the main step of GenVoca model.

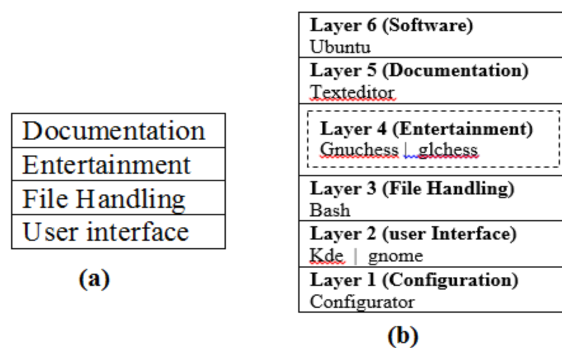

Fig. 9 GenVoca layer architecture of our example in Fig. 8.

Layers defined in Fig. 9(a), represents the functionalities. These layers are structured on the basis of dependency by adopting the bottom-up approach. Independent functionalities are on the lower level. In Fig. 10(b) layers further refined by presenting them with features. In Fig. 9(b) we have added two more layers "Software" at top level and "Configuration" at the bottom.

Ubuntu on the top layer represents the complete software with different functionalities as per user selection from the feature model. *Configurator* contains the values of basic properties of features e.g. selected/rejected, relevance, cross tree constraints. In our example, this configuration is required by all layers in-order to perform further checks about the selection or rejection of features. The configuration values propagate upward to the top for all layers.

4.1.3 GenVoca Grammar

In GenVoca model, we select features in top-down approach. On each layer one feature will be selected to complete the product. To develop a product if we start selecting featured from layered architecture of our example, the first selection will be *ubuntu* and on next layer *texteditor* will be selected. The features on Entertainment layer are *gnuchess* or *glchess* that are optional. For the feature *texteditor,* we need *bash*. These features require *Kde* or *gnome* for user interface so, one of the possible instance is *ubuntu[texteditor[bash[Kde]]]*. On the basis of this selection procedure, we can define the grammar rules for our example. These grammar rules are helpful for the implementation of GenVoca layers. Our defined grammar rules are as follows

ubuntu: ubuntu [File Handling]
File Handling: Text Editor [texteditor] | texteditor
texteditor: texteditor [User Interface] | User Interface
File Handling: File Handling [bash] | bash
bash: bash [User Interface] | User Interface
Entertainment: gnuchess [User Interface] | glChess[User Interface] | Entertainment[ ]
User Interface: kde[configurator] | gnome [configurator]
Configurator: exclude, selected, relevance, parent,

Please note that our example of Fig. 8 contains *games* as optional feature. So, in grammar we defined entertainment with empty set. This represent that *games* may not be selected.

4.1.4 Implementation

In this section we present the proposed automation process for void feature detection by implementing the functionalities presented in GenVoca layers. In the proposed process, we develop high quality system for the

detection of void feature models. Main reason behind this effort is to detect the quality of a feature model automatically. As both errors are caused by the wrong cross-tree constraints so, we focused on this. The proposed automation process implemented in C++ by using object oriented approach.

- In our approach, every feature requires the values of feature's properties (i.e. exclude, selected, relevance and parent) that are on the lower layer.
- The selection or deselection of features depend on its own properties and the properties of features on the beneath layer.
- Properties required by all feature from bottom layer to the top layer so, we implanted these properties as *configurator*. This *configurator* propagates upward to complete the automation process. For this configuration we devised configurator class that is presented in Fig. 11. We used objects of this configurator class to set the properties of features and to pass the values to the upper layer as parameter.
- The configurator class not only collects the values of basic properties and identifies the contradiction of features due to imply and exclude constraints.
- Here a question arise "If we are implementing proposed technique by considering feature models as case study then why we are getting input from users?". The answer is, to evaluate the proposed technique (explained in section 5. Evaluation).

```
class configurator {
public:
 bool selected;
 string relevance;
 string parent;
 int total_exclude, total_require, total_required_by;
 char exclude[50];
 string require[50];

 void config(){

     wcout <<"\nWhat is the relevance of Feature?   (R=Root, M=Mandatory, O=Optional, A=Alternative, X=OR) ";
     getline( cin, relevance);
     if(relevance !="R")
     {
          wcout <<"\nEnter the Name of Parent feature or press enter:-";
          getline( cin, parent);
     }
     wcout <<"\nHow many features exclude this feature?    ";
     cin>> total_exclude;
     for (int i=1; i<=total_exclude; i++)
     {
        wcout <<"Enter relevance of feature that exclude this one(M=Mandatory, O=Optional, A=Alternative, R=OR) "
       cin>>exclude[i];
     }
     wcout <<"How many features required by this feature?    ";
     cin>> total_require;
     if(total_require>=1)
```

```
        {
            for (int i=0; i<=total_require; i++)
            {
                wcout <<"\nEnter the name of feature that is required by this one:- ";
                getline( cin, require[i]);
            }
        }
            if(total_exclude>=1)
            {
                    for (int i=1; i<=total_exclude; i++)
                    {
                if ((exclude[i] == 'M') || (exclude[i] == 'R'))
                    selected =false;
            }
            }

        if ((selected != true) && (selected != false))
            selected=true;
            }

    void required_feature( configurator config_require)
  {
      if (config_require.selected == false)
       selected =false;
    }
};
```

Fig. 10 Implementation of *configurator*

After the implementing configurator, we consider the functionalities by implementing features. Our main is task to detect the inconsistencies in a given feature model, these inconsistencies are caused by contradictory cross tree constraints. So, in the implementation of features, we focused on the detection of cross-tree constraints. We start implementation from the lower layer to the upper layer. Hence, we implement features i.e. Kde and gnome, for the functionality of the user interface. We use structures for the implementation in C++. The implementation of *gui* feature implanted by the following structure.

```
1.struct gui{
2.              wcout <<"\n\n\t ** Enter the configurrtion of Feature gui ******\n"
3.    config_gui.config();
4.             if ((config_gui.total_require>=1) && (config_gui.selected==true))
5.                {
                   // to check that required feature is selected or not
6.              for (int i=1; i<=config_gui.total_require; i++)
7.                 {
8.          if((config_gui.require[i]=="bash") && (config_bash.selected==false))
9.              config_gui.selected=false;
```

```
10.         if((config_gui.require[i]=="games") && (config_games.selected==false))
11.                config_gui.selected=false;
12.         if((config_gui.require[i]=="gnuchess") && (config_gnuchess.selected==false))
13.                config_gui.selected=false;
14.                       if((config_gui.require[i]=="glchess")&&(config_glchess.selected=false))
15.                config_gui.selected=false;
16.                       if((config_gui.require[i]=="texteditor")&& (config_texteditor.selected==false))
17.                config_gui.selected=false;
18.                       if((config_gui.require[i]=="ubuntu") && (config_ubuntu.selected==false))
19.                config_gui.selected=false;
20                    }
21.       }
22.         if (config_gui.selected==true)
23.       {
24.           wcout <<"\n gui selected";
25.           selected_features[0]="gui";
26.           total_selected++;
27.       }
28.       else
29.         wcout <<"\n gui not selected";
30. };
```

Fig. 11 Implementation of *gui* feature.

Fig. 11 depicts the implementation of *gui* feature. In first statement, we define the structure for this feature with the name of *gui*. Statement number 2 is to display message for user. Statement number 3 is to call a method of configurator class. This method is to get values of basic properties of features and to check the contradictory cross tree constraints. Statement number 4 is to check the implies constraint by *gui* feature. On the same statement we are checking that *gui* is selected or not because if this a feature not selected then no need to move further. If condition on Statement number 4 is true, it means that *gui* is selected by the configurator and also have implies constraint on other feature. In Statement number 6 we are iterating for the number of required features. Statements 8 to 19 are to check that implied feature is selected or not. If implied feature not selected then *gui* will be deselected by setting the value of selected property to false. Statement number 22 is to detect that after performing all checks, *gui* is selected or not. If selected then on Statement 24 a message is being displayed and on Statement 25 *gui* added in the list of selected features. Statement number 26 counts the total selected features in given feature model. If *gui* is not selected then Statement 29 displays the respective message.

Remaining features also implemented in similar fashion, finally it is clear that which feature is selected. On the basis of selection or rejection we can decide about the void feature model error. Following statements are to check void feature model error of our example in Figure 8 on the bases of features properties.

```
1.   wcout <<"\n\n\t ********* Rsult ********\n";
2.   if(total_selected>=2)
3.   {
4.          wcout <<" \n Following features selected";
5.          for (int i=0; i<= total_selected; i++)
```

```
6.              cout<<'\n'<< selected_features[i];
7.    }
8.    else
9.          wcout <<"\n\t IT IS A VOID FEATURE MODEL";
```

Fig. 12 Statements to detect void FM error.

Statements in Fig. 12 are to decide whether feature model in our example is void or not, on the basis of selection or rejection of features. Statement 1 is to display the heading. Statement 2 is to check if numbers of selected features are more than one then it is not a void feature model. If condition on statement 2 is true then statements 4 to 6 will display the selected features. If condition is false it means only one (root feature) or no feature selected hence, it is void feature.

4.2 GenVoca model invalid instance

The following feature model contains invalid instance error because the mandatory features not selectable (Segura et al., 2010). We use this feature model as example for the implementation by GenVoca model.

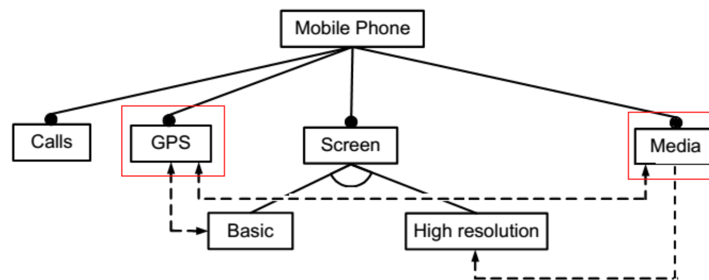

Fig. 13 Feature model with invalid product (Benavides et al., 2010).

Feature model shown in Fig. 13 contains the error of invalid instance because two mandatory features *GPS* and *Media* cannot be selected due to exclusion constraint between them. To get a valid product all mandatory features should be chosen in an instance (Segura et al., 2010).

4.2.1 Identifying and categorizing the components

We identify and categorize the functionalities as follows

Table 2 Functionalities of feature model in Fig. 13.

Connect	**Communication**	**Display**	**Entertainment**
Calls	GPS	Basic	Media
		High resolution	

The first functionality is "Connect" will be done by the feature *Calls*. For "Communication" functionality we have *GPS* feature. For "Display" there are *Basic* and *High resolution* alternatively. For "Entertainment" we have feature, *Media*.

4.2.2 Functional dependencies between Components

To model the GenVoca architecture layers, we need to find dependencies between features on the bases of general working. The following are the dependencies between features

Calls depends on *Screen*
GPS depends on *Screen*
Media depends on *Screen*

After finding the dependencies, we present the functionalities in layers.

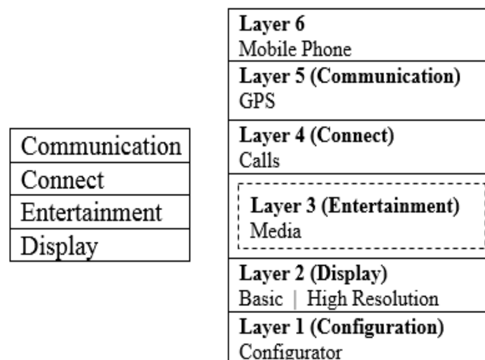

Fig. 14 GenVoca architecture of our example in Fig. 13.

Dependencies based layers defined in Fig. 14(a) while 14(b) depicts the layers with features.

4.2.3 GenVoca grammar

For the implementation, we define grammar rules of our example on the basis of top-down selection. Our defined grammar rules are as follows

Mobile Phone: Mobile Phone [Connect]
Connect: Connect [Calls] | Calls
Calls: Calls [Display] | Display
Communication: Communication [GPS] | GPS
GPS: GPS [Display] | Display
Entertainment: Entertainment[Media] | Entertainment []
GPS: GPS [Display] | Display
Display: Display[configurator]
Configurator: exclude, selected, relevance, parent,

Our example, depicted in Fig. 13 contains *Media* as optional feature. So, in grammar we define Entertainment with empty set. This represent that *Media* may not be selected.

4.2.4 Implementation

We are going to present the proposed automation process for invalid instance error detection by implementing the functionalities presented in layers. Inconsistency based errors occur due to wrong tree constraints so, we focused on this. The proposed automation process implemented in C++ by using object oriented approach.

It is implemented in similar fashion as we have done for the void feature model in section 4.1. We are using the same configurator to store the values of feature's properties that is already explained in Section (use number here).

```
1. struct Screen{
2.         wcout <<"\n Enter the configurrtion of Feature Screen \n";
3.         config_Screen.config();
```

```
4.                  if ((config_Screen.total_require>=1) && (config_Screen.selected==true))
5.                   {
                     // to check that required feature is selected or not
6.    for (int i=1; i<=config_Selected.total_require; i++)
7.                     {
8.          if((config_Screen.require[i]=="Calla") && (config_calls.selected== false))
9.                config_Screen.selected=false;
10.         if((config_Screen.require[i]=="GPS") && (config_GPS.selected== false))
11.               config_Screen.selected=false;
12.         if((config_Screen.require[i]=="Media") && (config_Media.selected==false))
13.         config_Screen.selected=false;
14.
15.                    }
16.            }

17.           if (config_Screen.selected==true)
18.           {
19.                  wcout <<"\n Screen selected";
20.                  selected_features[0]="Screen";
21.                  total_selected++;
22.            }
23.            else
24.             {
25.                  wcout <<"\n Screen not selected";
26.         if ((config_Screencvbfgh.relevance=="M") || (config_Screen.relevance =="m"))
27.         mandatory_not_selected++;
28.             }
29. };
```

Fig. 15 Implementation of *Screen* feature.

In Fig. 15, we present implementation of *Screen* feature. Statements 1 to 25 are same as explained in Figure 13. Statement 26 is to check the mandatory relevance of the features because if mandatory feature is not selected then this feature model will generate invalid instances. If condition being means that it is mandatory feature hence, Statement 27 is to calculate the mandatory features which are not selected. This will help to decide about invalid instances. If one or more than one mandatory features are not selected then this feature model will generate invalid instances.

5 Validation of Proposed Technique

The proposed technique automates the detection of inconsistencies in the feature models, so it should work accurately. To get the required output, all components (structures and classes) are tested separately and then joining them as a single software product. For testing, all possible combinations of instances with cross-tree constraints are entered. To test the technique, we implemented configurator that gets input of feature's properties from the user. User can enter any values of the properties of features from the feature models depicted in Figs 9 and 14. If a user enters the same values of feature's properties with same cross-tree

constraints, the result will be the required error but if the inputted values of features properties changed then output will be changed accordingly.

6 Conclusion

The feature models are used to develop the products so the quality of feature model is very important. If the feature model contains any kind of deficiency then the resulting product will be of low quality. One of the main reasons of deficiency in feature model is existence of errors. Inconsistency based errors are very harmful so, they degrade the quality of a feature model.

Our technique automate the detection of inconsistency based errors by adopting the state of the art model i.e. GenVoca layered architecture. By using the GenVoca model, we tried to detect both void feature model and invalid instance errors. We adopted the steps of GenVoca model for implementation. On first step, we identified and categorized the features on the basis of functionalities of the selected example. In second step, we define the dependencies of features. These dependencies are on functionality basis. When dependencies are listed, we stack the functionalities in layers. The layered architecture is in bottom-up approach. The independent functionality of feature model is on bottom layer and dependent on top layers.

In GenVoca layered architecture, each layer gets some parameter from the layer beneath it. This layered architecture further refined and each of the functionality presented with concern features. In refined layered architecture, we also added two more layer configurator in bottom and with complete product on top most layer. Configurator contains the configuration information of all features so, propagate upward to the top. Then we defined the GenVoca grammar for our example on the basis of top-down feature selection procedure. Next one is the main and required step of GenVoca model is implementation. All components implemented in C++. All components implemented in bottom approach so, configurator implemented. In configurator we allowed user to enter the values to validate the technique. After configurator all features implemented to detect the inconsistencies.

References

Batory D, Benavides D, Antonio R. 2006. Automated analysis of feature models: challenges ahead. Communications of the ACM, 49(12): 45-47

Batory D. 2005. Feature models, grammars, and propositional formulas. Software Process Improvement and Practice, 1: 7-20

Benavides D, Segura S, Ruiz-Cortés A. 2010. Automated analysis of feature models 20 years later: A literature review. Information Systems, 35(6): 615-636

Benavides D. 2007. On the Automated Analysis of Software Product Lines Using Feature Models. Dissertation, Universidad de Sevilla, Spain

Böckle G, Van Der Linden F. 2005. Software Product Line Engineering Vol. 10 (Klaus Pohl, ed). Heidelberg, Springer, Germany

Clements P. 2001. Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co Inc, Boston, MA, USA

Czarnecki K, Eisenecker UW. 2000. Generative Programming Methods, Tools, and Applications. Addison Wesley, USA

Czarnecki K, et al. 2000. Generative Programming and Active libraries. 25-39, Springer Berlin Heidelberg, Germany

Czarnecki K, Eisenecker UW. 1999. Components and Generative Programming. In: Proceedings of the 7th

European Software Engineering Conference (Nierstrasz O, Lemoine M, eds). 2-19, Toulouse, France

Felfernig A, David B, Galindo J, Reinfrank F. 2013. Towards Anomaly Explanations in Feature Models. In: Proceedings of the 15th International Configuration Workshop (ConfWS-2013). 117-124

Javed M, Naeem M, Wahab HA. 2014. Towards the maturity model for feature oriented domain analysis, Computational Ecology and Software, 4(3): 170-182

Javed M, Naeem M, Wahab HA. 2015. Semantics of the Maturity Model for Feature Oriented Domain Analysis. Computational Ecology and Software, 5(1): 77-112

Kang K, et al. 1990. Feature-oriented domain analysis (FODA) feasibility study. Technical Report, Carnegie-Mellon University Pittsburg, SEI, USA

Lung. Chung-Horng, et al. 2010. Experience of building an architecture-based generator using GenVoca for distributed systems. Science of Computer Programming, 75: 672-688

Maßen T, Horst L. 2004. Deficiencies in feature models. In: Proceedings of the Workshop on Software Variability Management for Product Derivation - Towards Tool Support. Collocated with the 3rd International Software Product Line Conference (SPLC'04). Springer Berlin Heidelberg, 3154: 331-331

Naeem M. 2012. Matching of Service Feature Diagrams based on Linear Logic. Dissertation, Department of Computer Science, University of Leicester, UK

Rosso CD. 2006. Experiences of performance tuning software product family architectures using a scenario-driven approach. In: Proceedings of the 10th International Conference on Evaluation and Assessment in Software Engineering (EASE 2006). 30-39

Segura S, et al., 2010. FaMa Test Suite v1.2. Technical Report ISA-10-TR-0. 1-52, Applied Software Engineering Research Group, University of Seville, Spain,

Thomas E. 2008. SOA: Principles of Service Design (Vol. 1). Prentice Hall, USA

Trinidad P, et al. 2008. Automated error analysis for the agilization of feature modelling. Journal of Systems and Software, 81(6): 883-896