COMBINATORIAL TESTING FOR SOFTWARE PRODUCT LINE WITH NUMERICAL CONSTRAINT

Do Thi Bich Ngoc*, Nguyen Quynh Chi*
*Học viện Công nghệ Bưu chính Viễn thông

Abstract—Software product lines (SPL) allow companies to represent a set of similar products developed from common core components. Thus, companies can increase the range of products efficiently. SPL is often represented by feature models. This representation may generate a huge number of product variants, including invalid configurations. Thus, testing this huge number of products is time consuming and expensive. This paper aims to reduce invalid configuration by extending the feature models with numerical features and numerical constraints. Besides, the paper proposes a combinatorial testing method extending a feature model to reduce the number of test cases.

Keywords—feature model, numerical constraint, combinatorial testing, software product line, testing.

I. INTRODUCTION

Software product lines (SPLs) [15] allow companies to efficiently increase the range of products by representing similar products developed from common components and with some variations in functionality. Therefore, instead of developing a collection of similar products individually, we can mass-customize products by exploiting their commonalities and maximizing reusable variation through a product line. Thus, SPL brings benefits in terms of higher productivity, shorter time to market and cost reduction. SPL is often represented by a feature model (like a tree structure) with "feature" here is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [8]. However, this representation may generate a huge number of product variants, including invalid configurations due to limitation of logic constraints in feature models. Thus, it challenges in testing variability throughout the whole product line lifecycle. Therefore, the objective of testing SPL is to specify the smallest number of test cases in certain amount of time such that specific coverage criteria are satisfied (e.g., all two software feature interactions are tested) and that all test configurations are valid (i.e., all dependencies between features are satisfied). Given a huge number of configurations, this manual task is extremely tedious and unsystematic, leading often to insufficient test coverage and redundancy in test cases.

Focusing on these problems, this paper proposed an extending feature model by adding numerical features and numerical constraints. The extending feature model allows us represent SPLs and requirements easily. Thus, the invalid configuration will be reduced. Besides, to reduce the number of test cases efficiently, the paper proposed how to apply a well-known combinatorial testing method using flattening algorithm for SPLs.

Related works

There are several works focuses on test case generation for Software product lines or feature models [1, 3, 4, 5, 6, 9, 10, 13]. We next surveys three most related works.

CTE-XL tool [7, 11], based on a classification tree method, allows users to generate combinatorial and three-wise covering test sets, while handling constraints among input parameters. However, constraints are handled in a passive way, by checking generated test configurations and possibly refuting inconsistent combinations. This approach is insufficient for a larger number of variables.

The second related work is the paper of Oster et. al [13] where they also proposed a flattening algorithm for software product line (SPL). However, the feature model is used in [13] is the original one, thus, it cannot represent numerical features and numerical constraints.

In [4], the author proposed an extending feature model with numerical and numerical constraints. However, the feature model in [4] only restricts to and-node and xor-node. Also, [4] aimed to find all configurations, not combinatorial configurations.

II. MODELING SOFTWARE PRODUCT LINE USING FEATURE MODELS

2.1 Feature models

Feature modeling has been introduced by Kang [8] as a compact and hierarchical representation of products in a SPL. A *feature model* is a hierarchically arranged set of features. Relationships between a *parent* (or *compound*) feature and its *child* features (or *sub-features*) are categorized as:

- *Alternative:* only one sub-feature can be selected,
- *Or*: one or more can be selected,
- *Mandatory*: features which are required,
- *Optional:* features which are optional.

Besides, to capture all domain restrictions, constraints between features (i.e., a feature ***requires*** another feature or two features are mutually ***exclusive***) have been added to complete the semantics of the models.

We call a SPL test configuration is one valid configuration of a feature model. This configuration is then used to form a test case. In the following, we will simply refer to SPL test configuration as ‘‘test configuration’’.

Valid/Invalid t-Tuple: A t-Tuple (where t is a natural integer giving the number of features presenting in the t-Tuple of features is said to be valid (respectively invalid), if it is possible (respectively impossible) to derive a product that contains the pair (t-Tuple) while satisfying the feature model's constraints.

Example:

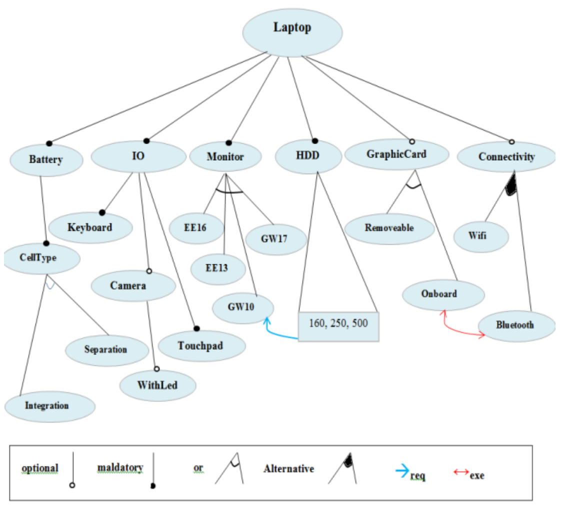


Figure 2.1 is a feature model of a SPL laptop.

We have feature types:

Mandatory: feature "*HDD*" must be selected in all laptop.

- **Optional**: feature "*Connectivity*" may be selected or not in a laptop.
- **Alternative**: feature "*HDD*" must be selected only value of *"160", "250", "500"* (i*.e., 160GB, 250GB* or *500GB*)
- **Or**: feature *"Connectivity"* can be selected as "*Bluetooth" or "Wifi"* or both.

We have constraints:

- **Requires**: if a laptop's "*HDD*" is "*160*" then the "*Monitor*" must be "*GW10"* .
- **Excludes**: there is no laptop with *"Connectivity"* being "*Bluetooth*" and "*GraphicCard*" being "*Onboard*".

2.2 Numerical feature models

Feature model (FM) allows only boolean features. However, there are several real feature models using numerical values. FM in Figure 2.1 is the first example. It is a part of a real feature model that represents Dell laptops from [16]. The feature model contains many features, in that several features are numeric. For example, "Monitor" (e.g., 10", 13", 16", 17"), "Hard Drive" (e.g., 160GB, 240GB, 500GB). Unluckily, these features are represented by strings, not numbers.

Another real example is the feature model for Trek Bikes from [16]. The feature model contains 543 features in that the feature Price is the parent of several features (e.g., 1001-2000, 2001 - 3000,..) and the feature Size is the parent of more than 30 features (e.g., 13", 14.5",...). We can see these features be numeric but they must be represented as string.

The string presentation of numerical features will restrict the constraints to boolean way only. It is a big limitation seen there are several contraints are numerical constraints (e.g. list all laptops with price < 800 and Monitor < 12.1")

We have some observations as the followings:

- To obtain a concrete model, we need to solve numerical constraints manually.
- Some abstract models are invalid since current logical constraints (i.e., **requires**, **excludes**) cannot represent the numerical constraints.

To solve these two problems we propose a new feature model in that numeric features and complex constraints are allowed. The type of features now can be boolean (as original FM) or numeric (e.g., integer, floating point...). For example, feature "HDD" in Figure 2.1 now have values set {160, 250, 500}. Besides, we now can also add numerical constraints besides logical constraint "req" and "excludes". For examples, we can add constraint: IF feature "HDD"<200 THEN feature "Monitor" <11.

III. COMBINATORIAL TESTING FOR NUMERICAL FEATURE MODELS

3.1 Combinatorial testing

Combinatorial testing (CT) [2, 12, 14, 15] is a testing technique, used to test interactions between parameter values. The effectiveness of CT is based on the observation that software failures are often due to interactions between only few (t) software parameters. A t-way testing covers all t-way combinations of input parameters and can detect faults caused by interactions of *t* or less components. The most often used CT application in practice is pairwise or 2-way testing. Pairwise testing requires that every pair of values is presented at least once in a set of test configurations. It was shown to be both time efficient and effective for most real case studies [12].

Combinatorial test designs support the construction of test cases by identifying various levels of combining input values for the assets under test. This approach is based on a simple process:

- Identify attributes that should vary from one test to another.
- For each factor, identify the set of possible values that the factor may have.

Apply a combinatorial design algorithm to cover all possible combinations of variants.

3.1 A flattening algorithm for the numerical feature model

The feature model is a (multi levels) tree structure while the input is required as a set of parameters and each parameter has a set of values. It can be seen as a simple (1 level) tree with only a root and a set of group childrens (values). Thus, to generate the test cases for the feature model, we need:

- Flatten the model from multi levels to 1 level (including only root and its children). In that, the correctness is guaranted by introducing contraints to ensure the test cases generated by original model are as same as test cases generated by flattening model.
- From the flattening model, generate the corresponding paremeters and values, constraints for combinatorial algorithms
- Apply combinatorial algorithms to generate combinatorial test cases.

In that, flattening is the most important step.

Flattening method:

Flattening method includes two main steps:

Step 1*:* all features and corresponding constraints will be lifted to the parent level. The process will stop until when all features become the children of root. The features then become parameters of combinatorial algorithm.

Step 2*:* Assign numerical values for these parameters.

There are several flattening rules to control the lifting step. These rules will be applied recursively until we obtain a feature model with two levels: root and its children. To ensure the semantics (i.e., set of products generated by original model is as same as set of products generated by flattening model) we sometimes need to add extra constraints.

Flattening rules:

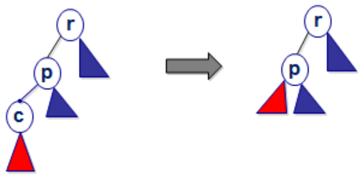
We need to create rule to lift the tree based on feature types of pair (parent, children). We consider following rules:

Rule 1. c is Mandatory, p is one of (Mandatory, Optional, Alternative, Or)

Because *c* is *Mandatory*, we cannot remove *c*’s children from tree. We then lift group children of *c* upto group children of *p*. To ensure the semantics, we must change *c* to *p* in constraint formulas. Thus, we must add *c*’s set of values to *p*’s set of values.

We have Rule 1:

- Lift group children of *c* up to *p*
- Remove *c* from the children list of *p*
- Change *c* to *p* in constraint formula
- Adding *c*’s set of values to *p*’s set of values



Example:

For the feature model in Figure 3.1, “Celltype” is *Mandatory*. It has two children: *Alternative* group “Integration”, “Separation”. Thus, applying Rule 1, we lift “Integration”, “Separation” to children of “Battery”. Then, we remove “Celltype” from the tree.

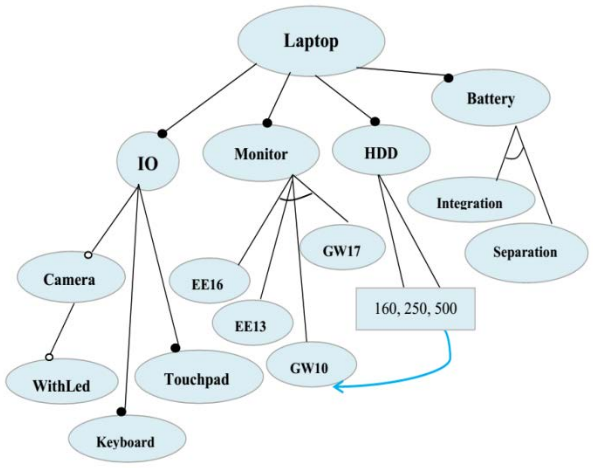


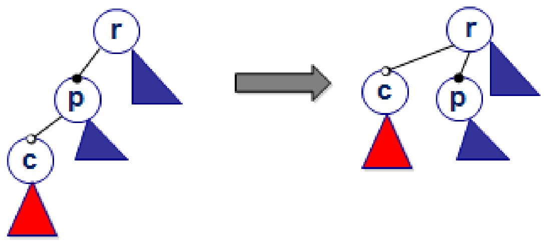
Figure 3.1 FM after applying Rule 1

Rule 2. c is Optional, p is Mandatory

Because *c* is *Optional* and *p* is *Mandatory*, we can lift *c* to be child of *r without losing the semantic*.

We have Rule 2:

- lift c to be child of *r*



Example:

In Figure 3.3, because “Camera” is *Optional*, “IO” is *Mandatory*, we can lift “Camera” to child of “Laptop”. Figure 3.4 shows the Laptop model after applying Rule 1, Rule 2.

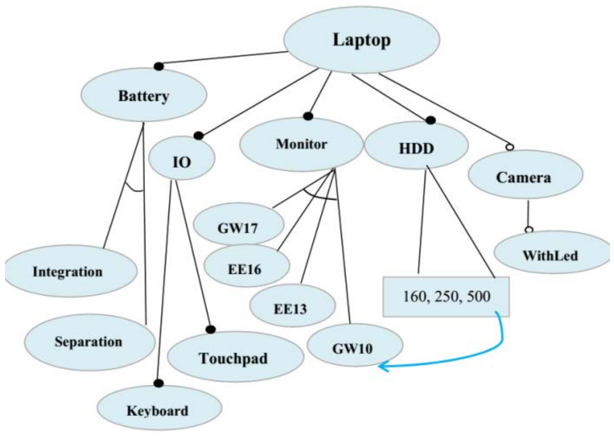


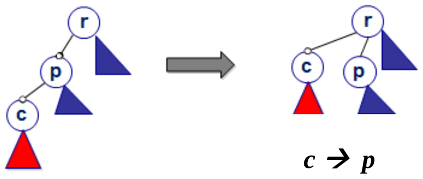
Figure 3.2 FM after applying Rule 2

Rule 3. c is Optional, p is one of {Optional, Alternative, Or}

Because *c* is *Optional*, we can lift *c* to be child of *r*. Beside, to ensure the semantic (i.e., if *c* is selected, then *p* must be selected), we add constraints: "c → p".

We have Rule 3:

- lift *c* to be child of *r*
- add constraint: *c* → *p*



Example:

In Figure 3.4, because "Withled" is *Optional* and is child of "Camera" ("Camera" is also *Optional*), thus, applying Rule 3, we lift "Withled" to be child of "Laptop", we also add a constraint: IF ("Laptop" == "Camera") THEN ("Laptop" == "WithLed").

Figure 3.5 shows the result of applying Rule 3 for Laptop model in Figure 3.4.

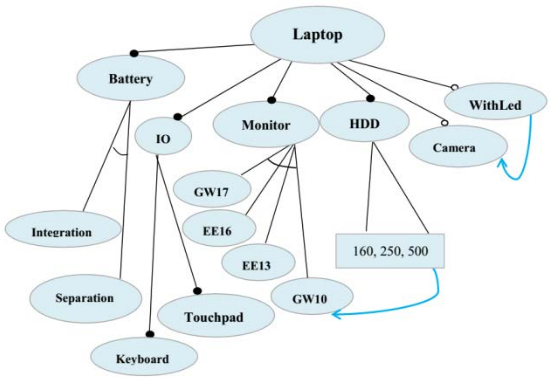


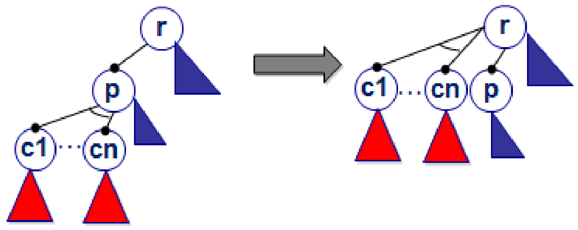
Figure 3.3 FM after applying Rule 3

Rule 4. (c_1,…, c_n) is Alternative group, p is Mandatory

Because *(c_1,…, c_n)* is *Alternative group* and *p* is *Mandatory*, we can lift *(c_1, …, c_n)* to child of *r without losing the semantic.*

We have Rule 4:

- Lift *(c_1, …, c_n)* to be *Alternative group* of *r*



Example:

In Figure 3.5, "Battery" and "Monitor" are *Mandatory*. "Integration", "Separation" are *Alternative group* and are children of "Battery". Thus, we lift "Integration", "Separation" to children of "Laptop". Similarly, we lift "EE16", "EE13", "GW17", "GW13" to be children of "Laptop", Figure 3.6 is Laptop model after applying Rule 4.

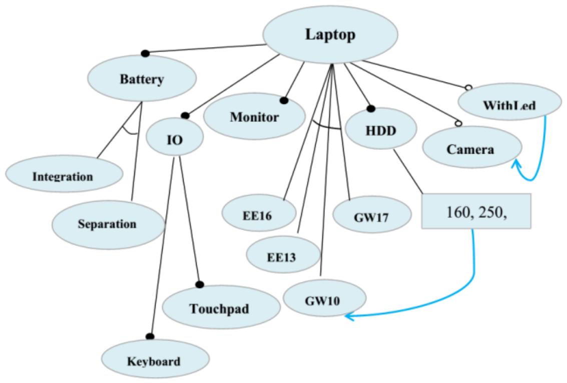


Figure 3.4 FM after applying Rule 4

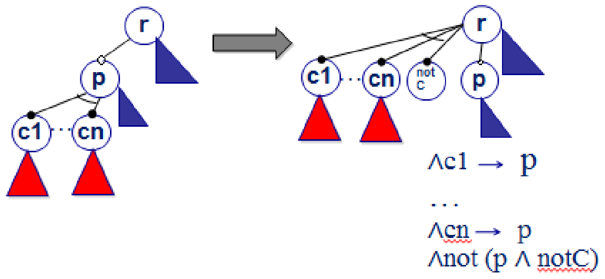
Rule 5. (c1,…, cn) is Alternative group, p is one of {Optional, Alternative, Or}

Because *(c_1,…, c_n)* is *Alternative group*, we can lift *(c_1, …, c_n)* to be child of *p*. To ensure the semantics, we add a new node, called "Notc", to *Alternative group (c_1, …, c_n, Notc)* and add constraint: *c_i → p.*

We have Rule 5:

- Lift *(c_1,…, c_n)* to be *Alternative group* chilren of *r*
- Add "Notc" to *Alternative group (c_1,…, c_n, Notc)*
- Add constraints:

c_1 → p;
c_2 → p;
…
c_n → p;
not (p **and** notc);



Example:

In Figure 3.5, "Removeable", "Onboard" are *Alternative group*, they are children of "GraphicCard" (*Optional*). Thus, we lift "Removeable", "Onboard" to *Alternative group* children of "GraphicCard". Then, we add "NotGraphicCard" to this *Alternative group* children. Besides, we add constraints:

IF (Laptop = "Onboard") THEN (Laptop = "GraphicCard";

IF (Laptop = "Removeable") THEN (Laptop = "GraphicCard";

NOT (Laptop = "NotGraphicCard" AND Laptop = "GraphicCard");

Figure 3.6 is Laptop model after applying Rule 5.

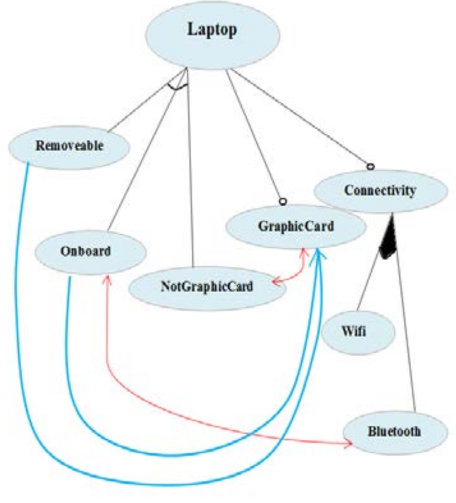


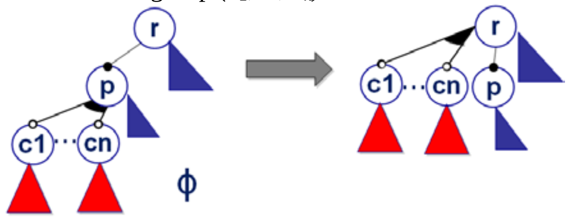
Figure 3.5: FM after applying Rule 5

Rule 6. (c1,…,cn) is Or group, p is Mandatory

Because $(c_1,\ldots, c_n)$ is *Or group* and *p* is *Mandatory*, we can lift $(c_1,\ldots, c_n)$ to be children of *r*.

We have Rule 6:

- Lift or group $(c_1,\ldots, c_n)$ to be children of *r*



Rule 7. (c1, …, cn) is Or group, p is one of {Optional, Alternative, Or}

Because $(c_1, \ldots, c_n)$ is *Or group* and *p* is *Optional*, we can lift $(c_1, \ldots, c_n)$ to be children of *r*. To ensure the semantics, we add a new node, called "Notc", to Or group $(c_1, \ldots, c_n, Notc)$ and add constraint: $c_i \rightarrow p$.

We have Rule 7:

- Lift children $(c_1, \ldots, c_n)$ to be *Or group* children of *r*

- Add "Notc" to *Alternative group* $(c_1, \ldots, c_n, Notc)$

- Add constraints:

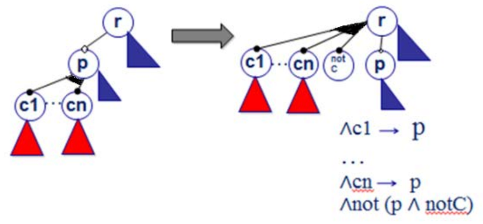
$c_1 \rightarrow$ p;

$c_2 \rightarrow$ p;

…

$c_n \rightarrow$ p;

not (p **and** notc);



Example:

In Figure 3.6, "Wifi", "Bluetooth" are *Or group* of *Connectivity* (*Optional*). Thus, we lift "Wifi", "Bluetooth" to be children of "Connectivity". To ensure semantic, we add node "NotConnectivity" to this *Or group*. We add constraints:

IF (Laptop = "Wifi") THEN (Laptop = "Connectivity");

IF (Laptop = "Bluetooth") THEN (Laptop = "Connectivity");

NOT (Laptop = "Connectivity" AND Laptop = "NotConnectivity");

Figure 3.7 is Laptop model after applying Rule 7.

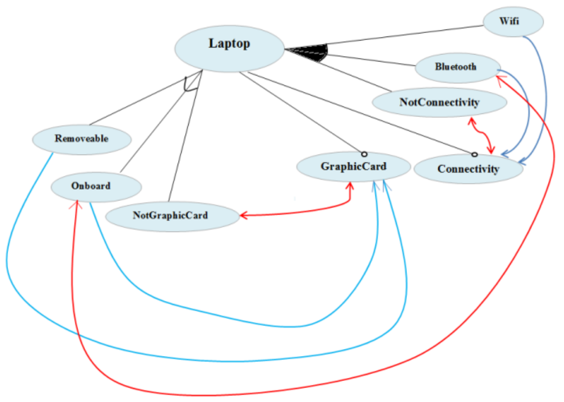


Figure 3.6: FM after applying Rule 7

Finally, the Laptop model after flattening (applying 7 rules) is shown in Figure 3.8. It is a one level tree with only one root and the list of children.

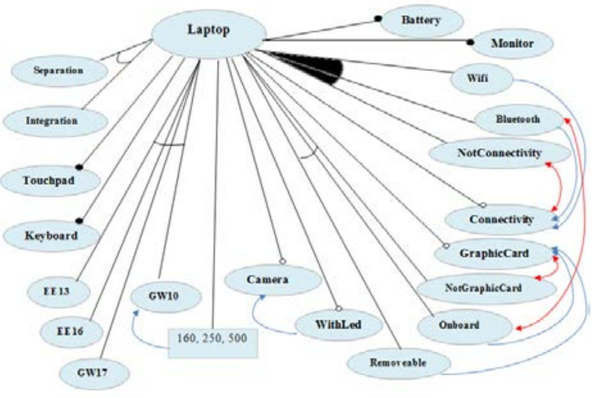


Figure 3.8: Laptop model after flattening

3.3 Generating Input for combinational testing tools

After flattening, we obtain a model (tree) has only one root *r* and set of group children p_i. We now transform it to the input format of combinational testing tools (e.g. PICT[2], ACTS[14]). Then, applying the corresponding tools will generate combinational test cases of SPL. We call:

- Boolean values are {0,1}: { 0 = true, 1 = false}
- $-\infty$ is the smallest value that computer can represent.
- *r* is the root, p is parent node, and c is p's child.

We have transforming rules:

1. p is mandatory:

Let <value_i> be a value of p, we have :

p: < value_1 >, < value_2 >,….< value_n >

2. p is optional:

Let <value_i> be a value of p, we have :

p: <value_1>, <value_2>,….<value_n>, <$-\infty$>

3. (p_1,…,p_n) is alternative group:

Let < value_{ij} > be value of (p_i), i in [1,n], we have :

P: < value_{11} >, < value_{12} >,….< value_{nm} >

4. (p_1,…,p_n) is or group:

Let < value_{ij} > be value of (p_i), i in [1,n], we have :

p_1: <value_{11}>, <value_{12}>,….<value_{1m}>, <$-\infty$>

…

p_n: <value_{n1}>, <value_{n2}>,….<value_{nm}>, <$-\infty$>

We add constraints:

NOT (([p_1] = < $-\infty$ >) AND … AND ([p_n] = < $-\infty$ >))

We also need to edit the constraints as follow:

- If p is Boolean:
  + Change p by ([p] = 0) in logic constraints
  + Change p by [p] in numerical constraints
- If p is numeric:
  + Change p by ([p] in {<value_1>,…<value_n>}) in logic contraints
  + Change p by [p] in numerical constraints
- If the constraint is a → b: replace by IF a THEN b;

IV. EXPERIMENTS

We do experiments for several feature models. The Table 4.1 shows the experiments with several product lines (i.e., Volume product line, Computer Hardware configuration product line, Computer software product, TV product line, and Laptop product line in Figure 2.1) based on real feature models from [16]. "Feature model" column shows the list of SPL; "Number of features" column shows the number of features in the corresponding SPL; "Number of constrains" shows the numerical constraints appearing in the corresponding SPL; "Number of combinatorial test cases" column shows the number of test cases with constraints and without constraints.

Table 4.1: Experimental results

Feature model	*Number of features*	*Number of constraints*	*Number of combinatorial testing*	
			No constraints	*Use constraint*
Volume product line	*36*	*2*	*4704*	*62*
Computer hardware product line	*34*	*2*	*480*	*29*
Computer software product line	*24*	*1*	*216*	*14*
TV product line	*15*	*1*	*97*	*10*
Laptop product line	*25*	*2*	*1728*	*32*

The experiments show that:

- Applying combinatorial testing reduces a huge number of test cases
- Extending feature model with numerical features and constraints allows us represent more flexible and more efficient specification.

The proposed method can be applied for real SPLs.

V. CONCLUSIONS

There are two main challenges for making SPL testing practical. First, tests often contain invalid product configurations that cause failures in execution. Second, there is a lack of measurable test coverage criteria. In this work, we provide a method that addresses each of these limitations. The method (1) allows automated checking for validity of test configurations, (3) leverages combinatorial testing to increase test coverage.

To automatically generate valid test configuration, this paper proposed an extending feature models by adding: (1) numerical features instead (the original feature model allows only Boolean feature); (2) numerical constraints (the original feature model only allows only logical constraints with requires and mutex). The proposed feature model allows us represent feature models and requirements more effectively and easily.

To obtain test coverage, the paper also proposed a method to generate combinatorial testing automatically by using a flattening algorithm. This combinatorial testing method will increase test coverage.

In future, the proposed method can be improved and extended by adding other black-box testing techniques (e.g., boundary testing). Another direction is to apply the proposed method to the real world.

REFERENCES

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

[2] Czerwonka, J., Pairwise testing in the real world. practical extensions to test case generators. Microsoft Corporation, Software Testing Technical Articles, 2008.

[3] Do, T. B. N., Kitamura, T., Nguyen, V. T., Hatayama, G., Sakugari, S., and Ohsaki, H. - Constructing test cases

for n-wise testing from tree-based test models, In Proceedings of the 4th International Symposium on Information and Communication Technologies, 2013, pp. 275–284.

[4] Do T. B. N., -Numerical feature-tree for testing, Tạp chí khoa học và công nghệ, tập 52-số 6C, 2014 pp.57-67.

[5] Dusica M., Arnaud G., Sagar S., Aymeric H. -Practical Pairwise Testing for Software Product Lines. SPLC 2013, Aug 2013, Tokyo, Japan.

[6] Gilles P., Sabastian O., Sagar S., Jacques K., Benoit B., et al. -PairwiseTesting for Software Product Lines: Comparison of Two Approaches. Software Quality Journal, Springer Verlag (Germany), 2012.

[7] Grochtmann, M., Wegener, J., & Grimm, K. -Test case design using classification trees and the classification-tree editor CTE, In Proceedings of Quality Week (Vol. 95, p. 30), 1988.

[8] Kang, K. C., Cohen, S. G., Hess , J. A., Novak, W. E., and Peterson , A. S.. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.

[9] Kitamura, T., Do T. B. N., Ohsaki, H., Fang, L., and Yatabe, S. -Test-case design by feature trees, In Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, 2012, pp. 458–473.

[10] Kitamura T., Yamada A., Hatayama G., Artho C., Choi E.H., Do. T.B.N, Oiwa Y., Sakuragi S., -Combinatorial Testing for Tree-structured Test Models with Constraints", in Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS2015), pp. 141-150 IEEE CPS, 2015.

[11] Lehmann, E., and Wegener, J. -Test case design by means of the CTE XL, In Proceedings of the 8th European International Conference on Software Testing, Analysis & Review, Kopenhagen, Denmark, 2000.

[12] Richard Kuhn, Y. L. D., Raghu N. K., Practical Combinatorial Testing. Technical Report NIST/SP-800-142, National Institute of Standards and Technology, 2010.

[13] Oster, S., Markert, F., and Ritter, P.. Automated incremental pairwise testing of software product lines. In Proc. of SPLC'10, pages 196–210, 2010.

[14] ACTS, available from: http://csrc.nist.gov/groups/SNS/acts/index.html.

[15] Software Product Lines | Overview, available from: http://www.sei.cmu.edu/productlines/.

[16] Repository of Real Feature Models, available from: http://www.splot-research.org

Do Thi Bich Ngoc, received the PhD degree from Japan Advanced Institute of Science and Technology in 2007. Curently, she is lecture of Faculty of Information Technology in Posts and Telecommunications Institute of Technology. Her research intertests are software testing, formal methods, program analysis, and data mining.

Nguyen Quynh Chi, currently a lecturer of the Faculty of Information Technology at Posts and Telecommunications Institute of Technology in Vietnam. She received a B.Sc.in Information Technology in Hanoi University of Technology in Vietnam, a M.Sc. in Computer Science in University of California, Davis, USA (UCD) and became PH.D Candidate at UCD in 1999, 2004 and 2006, respectively. Her research interests include machine learning, data mining, and testing algorithms.