Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 314 (2015) 61–81

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Method to Identify Corrections of Defects on Product Line Models

L. Rincón [a,1], G. Giraldo [b,2], R. Mazo [c,3], C. Salinesi [c,3], D. Diaz [c,3]

[a] *Departamento de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana, Cali, Colombia*

[b] *Departamento de Ciencias de la Computación y de la Decisión,Universidad Nacional de Colombia,Medellín,Colombia*

[c] *CRI,Panthéon Sorbonne University, Paris, France*

Abstract

Software product line engineering is a promising paradigm for developing software intensive systems. Among their proven benefits are reduced time to market, better asset reuse and improved software quality. To achieve this, the collection of products of the product line are specified by means of product line models. Feature Models (FMs) are a common notation to represent product lines that express the set of feature combinations that software products can have. Experience shows that these models can have defects. Defects in FMs be inherited to the products configured from these models. Consequently, defects must be early identified and corrected. Several works reported in scientific literature, deal with identification of defects in FMs. However, only few of these proposals are able to explain how to fix defects, and only some corrections are suggested. This paper proposes a new method to detect all possible corrections from a defective product line model. The originality of the contribution is that corrections can be found when the method systematically eliminates dependencies from the FMs. The proposed method was applied on 78 distinct FMs with sizes up to 120 dependencies. Evaluation indicates that the method proposed in this paper scale up, is accurate, and sometimes useful in real scenarios.

Keywords: Software product lines, Features Models, Corrections, Defects, Software Engineering

1 Introduction

Product line engineering is a promising production approach used to manage in an efficient way a set of products that belong to a particular domain and have common and variable elements. This approach offers benefits such as reduced time to market, increased reuse and increased quality [29]. Benefits obtained with Product Lines (PL) are extensible to software engineering, due to in the software development area

[1] Email:{lfrincon}@javerianacali.edu.co

[2] Email:{glgiraldog}@unal.edu.co

[3] Email:{raul.mazo,camille.salinesi,daniel.diaz}@univ-paris1.fr

http://dx.doi.org/10.1016/j.entcs.2015.05.005

is also necessary to manage reuse and variability. Specifically, in the context of the software engineering, product lines are named Software Product Lines (SPL) [5].

Product line engineering represents in an intensive way all valid products belonging to a PL by means of product line models. In this sense, the feature models (FMs) is one of the available notations to represent product line models. FMs are designed during the early stages of the PL development, and they are a key input to identify common and variable elements of the PL [12]. In a FM, each feature is a prominent or distinctive user-visible aspect of a software system. Thus, FMs are useful to communicate effectively with customers and other stakeholders such as marketing representatives, managers, production engineers, system architects, etc. [12].

Having FMs that correctly represent the domain of the product line is of paramount importance for product line engineering success. However, as FM complexity grows, semantic defects may be unintentionally introduced, which decreases the quality of the FM, and consequently the benefits from the product line. Specifically, semantic defects are imperfections that affect the ability of FM to represent all the desired products [35].

The literature provides several approaches to automatically identify semantic defects in FMs [4, 25, 26, 35, 39, 40, 42–45]. However, only a few of these proposals are able to explain how to fix defects, and these approaches only find some of the possible corrections [25, 39, 40, 42, 44]. This means that once defects are found in a FM, it is necessary to manually inspect the model to detect available corrections. Nevertheless, this is a cumbersome task that depends on experience and skills of the model designer. Indeed, looking for the corrections of defects is almost as complicated as looking for defects themselves. In fact, the number of dependencies and interrelations among them make finding corrections an error-prone, tedious, and sometimes unfeasible task [26, 40, 45].

The general goal of our research is to find a generic technique that will point out the cause of various kinds of defects in product line models specified with different notations. In this paper, we propose a step towards this goal. Particularly, we present a new method that identifies defects in FMs, and detects possible corrections for each defect.

Specifically, the proposed contribution can be summarized as follows:

(i) A method that identifies potential corrections of defects in FMs.

(ii) We suggest to exploit *Minimal Correction Subsets (MCSes)* to detect corrections of defects in FMs. The concept of MCSes comes from the constraint programming area. To the best of our knowledge, it has never been used before for identifying corrections of defects in FMs.

(iii) An automated tool to implement our approach.

(iv) A preliminary evaluation was performed. It indicates that the proposed method is scalable, accurate and useful in real scenarios.

The remaining parts of this paper is structured as follows. Section 2, gives a brief overview of the necessary concepts for understanding the proposed contribution.

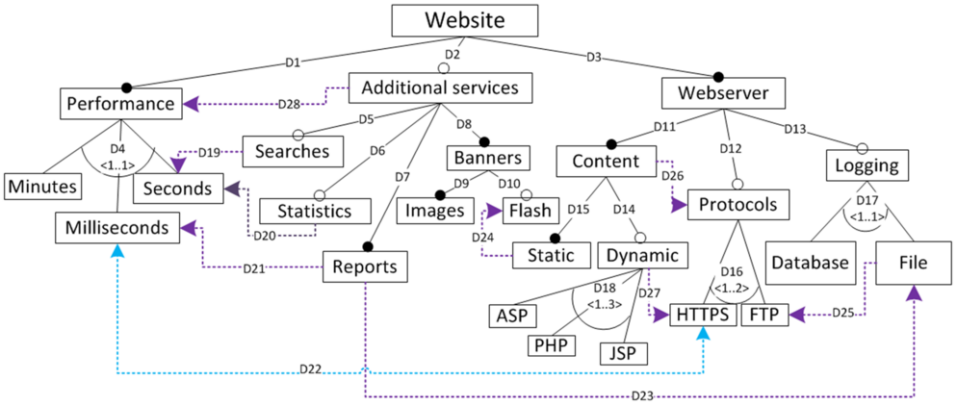

Figure 1. Website feature model. Adapted version of the website product line proposed by Mendoça *et al.* [22]

Section 3 explains our method to identify potential corrections of defects in FMs. Section 4, presents implementation details, and Section 5 shows the preliminary evaluation results. Section 6, provides a summary of related works, and Section 7 presents the conclusions and suggests future research directions.

2 Preliminary concepts

2.1 Feature models

Feature modelling is a notation used to represent product line models. Each feature is a node in a tree structure. The tree structure represents the hierarchical organization of the features where the root of the tree represents a whole product line, and therefore is part of all valid products of the product line. Furthermore, if a feature represented by a non-root node is selected in a product, its father feature is selected too. Figure 1 presents the product line model of a Web site product line that was specified using feature modeling. As the Figure 1 shows, features are interrelated with direct arcs named dependencies [12] and there are five types of dependencies: *mandatory, optional, group cardinality, requires* and *excludes*.

- **Mandatory:** dependencies are depicted by a line that connects a parent feature with a child feature through a line ending with a filled circle. These dependencies indicate that child features should be included in all valid products containing the parent feature and vice versa. For example, in Figure 1 the dependency *D1* connects the features *Website* and *Performance* with a mandatory dependency. Therefore, all products with the *Website* feature include the *Performance* feature too. In Figure 1 other mandatory dependencies are *D3, D7, D8, D9, D11* and *D15*.
- **Optional:** dependencies are depicted by a line that connects a parent feature with a child feature through a line with an empty circle at the end. These dependencies indicate that child feature may or may not be included in the valid products that contain the parent feature. Furthermore, if the child feature is

included in a product, then its father feature should be included too. For example, in Figure 1 the dependency *D2* is an optional dependency. Therefore, products that include the *Website* feature may or may not include the *Additional Services* feature. Figure 1 shows other 6 optional dependencies: *D5, D6, D10, D12, D13* and *D14*.

- **Group Cardinality:** dependencies represent an interval that limits the number of children features that can be included in a product when their parent feature is selected. For example, in Figure 1 the dependency *D18* is a group cardinality that requires minimum one and maximum three of its children features. Other group cardinalities dependencies in Figure 1 are *D4, D16* and *D17*.
- **Requires:** dependencies are depicted by a dotted line with a simple arrow at the rear end. These dependencies indicate that some feature (connected at the start of the edge) requires the presence of some other feature (at the end of the edge) in the same product. For example, in Figure 1, *D19* is a requires dependency, therefore the feature *Seconds* is required by the feature *Search*. Other requires dependencies in Figure 1 are *D20, D21, D23, D24, D25, D26* and *D27*.
- **Excludes:** dependencies are depicted by a dotted line with an arrow at the two ends. Features related by excludes dependencies cannot be included together in any valid product. For example, in Figure 1, *D22* is an excludes dependency, therefore no product will include features *Milliseconds* performamce and *HTTPS* protocol at the same time.

Mandatory and optional are structural dependencies, whereas requires and excludes are cross-tree dependencies [6].

2.2 Semantics defects in feature models

The semantics of FMs describes the collection of all possible products that can be derived from FMs [6]. In this paper, we are interested in semantic defects of FMs. These defects are imperfections that affect the ability of FMs to represent all desired products [35], and consequently adversely affect the semantics of the model. Next, we present the most common semantic defects found in FMs. Henceforth the paper simply refers to semantic defects as "defects".

- **Void models:** FM does not allow deriving any valid product [35, 40, 43].
- **False product line:** FM permits deriving one valid product only [35].
- **Dead features:** these features are not present in any valid product derived from the product line, although they are part of the FM [35, 43].
- **False optional features:** these features are declared as optional in the FM but they are present in all valid products derived from the product line [35, 40, 43].
- **Redundancies:** they are dependencies that do not change the semantics of the FM. Redundancies take place when FM has the same information modeled in different ways [35, 43]. Although there may be cases where the designer intentionally introduces redundancy in the FM. In this paper, all identified redundancies

are considered defects, because redundancies are a problem for the evolution of FMs [35].

2.3 Running example

The next of the paper uses as running example an adapted version of the website product line proposed by Mendoça *et al.* [22] (cf. Figure 1). This product line is a solution to rapidly develop new websites based on common and variable features shared among different websites.

The original model was intentionally simplified, and ten dead features (i.e. *ASP, Database, Searches, Dynamic, Statistics, HTTPS, JSP, Minutes, PHP, Seconds*), seven false optional features (i.e. *File, Flash, FTP, Milliseconds, Protocols, Additional services, Logging*), three redundancies (i.e. dependencies *D25q, D26 and D28*) were added to the model. The model was also turned into a false product line (no product can be derived from it). For the sake of simplification we only refer to the dead feature *ASP* in the rest of the paper.

Our adaptation of the website FM has three important features: *Performance, Additional services* and *Webserver* (cf. Figure 1). The *Performance*, according its select feature, is expressed in *Milliseconds, Seconds or Minutes*. *Additional Services* are *Searches* on the website, *Statistics* of visits, *Reports* and *Banners*, and the *Webserver* feature groups the *Content*, the transfer *Protocols* and the *Logging* capacity.

Banners are always *Images* and sometimes *Flash* animations. If a website has *Additional services*, it also has *Reports* and *Banners*. Frequently, the content of websites is *Static* but they could also include *Dynamic* content written in *PHP, JSP* or *ASP*. *HTTPS* and *FTP* are the supported protocols. Thus, for each website derived from this FM it is possible either to select one of these protocols or both at the same time. Optionally, derived websites include the *Logging* feature and include logs. Logs are useful to record who and what operations are performed on the website. Logs may be stored in *File* or in *Database*.

Other dependencies restrict the possible combinations of features for obtaining valid products in this FM. For example, if a *Website* saves its logs in *Files*, then it requires the *FTP* protocol (i.e. Dependency *D25*), and no product will support a performance in *Milliseconds* if *HTTPS* protocol is required (i.e. Dependency *D22*).

2.4 Constraint programming

Constraint programming is a paradigm for solving combinatorial search problems and is currently applying to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics [33]. The Constraint programming paradigm is useful to solve Constraint Satisfaction Problems (CSP), they are mathematical problems represented by variables and constraints where each variable has a domain of values, and each constraint delimits the possible values that the variables may take. A CSP is solved if all constraints are satisfied while, at the same time, each variable has assigned a single value of its domain [10]. Conversely, when

no solution is found the CSP is unsatisfiable [33]. Constraint Satisfaction Problems could be resolved with constraint solvers [4] such as GNU Prolog [7] or SWI Prolog [46]. Constraint programming has been used in the product line engineering to represent the semantics of FM in order to reason about them [19,21,35].

2.5 Minimal Corrections Subsets (MCSes)

A MCS is an irreducible subset of constraints whose removal makes solvable an unsolvable constraint program. This term comes from the constraint programming area, and it is usually used for detecting corrections of unsolvable constraint programs [3,15,24,28]. For example, MCSes have been successfully applied in hardware design verification task [1] and circuit diagnosis [34].

If there is more than one subset of corrections, then there are multiple MCSes [15]. Each MCS is minimal because it must be completely removed from the unsolvable constraint program in order to turn the program solvable. This ensures that each MCS has only relevant constraints to fix the unsolvable constraint program [15].

Formally,

$$\begin{gathered}\text{Given an unsatisfiable constraint program } \alpha \\ M \subseteq \alpha \text{ is a MCS} \equiv \alpha - M \text{ is satisfiable} \wedge \\ \forall Ci \in M,\ \alpha - (M - \{Ci\}),\ \text{is unsatisfiable.}\end{gathered}$$

MCSes are minimal but they do not necessarily have a maximal number of elements. For example, as we will explain in Subsection 3.3, we identify for the *ASP* feature MCSes with one, two and three elements.

3 Proposal

In this section, we describe our method to identify all types of defects presented in 2.2 and their respective corrections. Those corrections are minimal subsets of dependencies that should be removed from the FM in order to correct at least one defect (MCSes). To achieve that, the method receives a FM as input and carries out three steps: it transforms the FM (Subsection 3.1), identifies the defects of the transformed FM (Subsection 3.2), and finally identifies corrections for each detected defect (Subsection 3.3). This last step is our main contribution. At the end, the method presents the defects found and their related corrections.

3.1 Step 1. Transformation of feature models

The graphical representation of FMs does not allow to automatically reason about them. Therefore, the input FM are transformed into a new model named *Transformed model*. This *Transformed model* is useful, in the next steps of our proposal to create constraint programs on which we can automatically reason about the FM.

[4] A solver is a generic term indicating a piece of mathematical software that 'solves' a mathematical problem. A solver takes problem descriptions in some sort of generic form and calculates their solution [18].

Particularly, in the *Transformed model* we keep: (i) the features and dependencies of the FM, (ii) a description in natural language of each model dependency, and (iii) a representation with constraints of each dependency of the FM. We get each constraint using the transformation rules proposed by Mazo et *al.* [21]. In this way, when in the next steps we create constraint programs, each feature is represented like a boolean variable, and each dependency is represented like a constraint [21].

3.2 Step 2. Identification of defects

Salinesi and Mazo [35] propose a series of algorithms to identify the types of defects presented in Subsection 2.2 by analyzing models specified as constraint programs.

In this step, we transform the input FM into a constraint program using the *Transformed model*. Then, our method analyzes the resulting constraint program with the algorithms proposed by Salinesi and Mazo [35]. In this way, when this step ends the method has identified the defects of the analyzed FM and it is ready to identify the possible corrections of each defect.

3.3 Step 3. Identification of corrections

The method proposed in this paper uses MCSes to identify potential corrections of defects in FMs. Thus, identifying corrections of defects in FMs corresponds to identify the MCSes of an unsolvable constraint program. In terms of FMs, every MCS is a minimal subset of dependencies that should be removed from the FM in order to correct a defect.

To detect the MCSes, our method considers as input a transformed FM and the defect to analyze. Then, the method creates an unsolvable constraint program that is analyzed to identify the MCSes; i.e., the corrections. This step should be repeated for each defect in the FM.

The rest of this section presents four elements that will be used to identify the MCSes. The first element discusses the typology of constraints needed to identify the MCSes according to the type of defect. The second one is about the algorithm proposed (i.e., Algorithm 1) to identify MCSes. The third one deals with the procedure to obtain the corrections of each defect when the identified MCSes are transformed into dependencies of the FM to be analyzed. Finally, the fourth element presents the procedure to analyze the corrections identified by the proposed method. Specifically, we ilustrate how Algorithm 1 works for identifying the correction(s) of the *ASP* dead feature presented in Figure 1.

3.3.1 Types of constraints

In order to identify the corrections of each defect, Algorithm 1 which will be explained later systematically analyzes unsolvable constraint programs. The union of three types of constraints forms these constraint programs. These three types of constraints are: *constraints of the model*, *fixed constraints* and *verification constraints*.

- ***Constraints of the model:*** are the constraints that represent the dependencies

of the FM. Algorithm 1 removes these constraints from the transformed FM, in order to identify the MCSes of each defect.

- ***Fixed constraints:*** are constraints related to the notation in which the product line model is represented. A fixed constraint of a FM is for instance, that the root feature must be selected in any product derived from the FM [12].
- ***Verification constraints:*** represent the defect to analyze. This kind of constraints serves to make unsolvable a constraint program, since an unsolvable constraint program is the starting point for identifying corrections of defects in FM.

3.3.2 Verification constraint by type of defect

The verification constraint needed to identify the MCSes with Algorithm 1 changes according to the type of defect as follows:

- **Void model:** the corrections of a void model are the collection of constraints that should be eliminated to make solvable the corresponding constraint program. It is worth noting that to identify the corrections for this kind of defect, it is not necessary to add any *verification constraint* because, when the FM is void, the constraint program formed by the *constraints of the model* and the *fixed constraints* is already unsolvable.
- **Dead features and false optional features:** to identify corrections of a dead feature, our method identifies the constraints that by eliminating them make it possible to configure products containing this feature. Similarly, identifying corrections of a false optional feature consist in identifying the constraints that by eliminating them make it possible to configure products without this feature. In order to do so in the case of dead features, it is necessary to create a *verification constraint* that requires the selection of the "dead feature" on at least one product of the FM. In the case of false optional features, it is necessary to create a *verification constraint* requiring a product without the "false optional feature", i.e., a product in which the false optional feature is not selected. For example, `ASP #=1`, is the *verification constraint* that our method automatically adds to identify whether the feature *ASP* is dead or not. This restriction forces the *ASP* feature to be selected and makes unsolvable the resulting constraint program, i.e., the union of *the constraints of the model, the fixed constraints and the verification constraints.*
- **False product line:** a produt line model is false if it does not allow deriving more than one product [35]. This may be because the features model has no variability (all features are mandatory), or because the model has false optional or dead features. When dead and false optional features are fixed, more products can be configured from the corresponding product line model. Therefore, fixing dead and false optional features also corrects the defect related with false product line models. Otherwise, if the FM is a false product line model because all features are mandatory, our method does not identify any correction for this defect, based on the assumption that this was intentional.
- **Redundancies:** to identify corrections for redundant dependencies means iden-

tifying constraints that could be eliminated from the model without changing its semantics (a list of correct configurations). To achieve this, the *verification constraint* should represent the negation of the redundant constraint that we want to analyze. Thus, the union of *the constraints of the model, the fixed constraints and the verification constraints* results in an unsolvable constraint program.

3.3.3 Algorithm for Identifying Minimum Correction Subsets

This paper proposes an original algorithm to identify the MCSes of an unsolvable constraint program (cf. Algorithm 1). This algorithm identifies the MCSes by systematically removing the *constraints of the model* from an unsolvable constraint program until the resulting constraint program has at least one solution. Each constraint that makes solvable the resulting constraint program, belongs to a MCS once it is eliminated. To find all the MCSes, it is necessary to remove the *constraints of the model*, one by one, then two by two and so on.

This pass of the method creates an unsolvable constraint program that represents a FM with one or more defects, and executes Algorithm 1 for each defect identified at step 2 (see Subsection 3.2). The inputs of Algorithm 1 are the *constraints of the model* `(mc)`, the *fixed constraints* `(fc)`, and the *verification constraints* `(vc)`. As output, the algorithm delivers the collection of all the Minimum Correction Subsets (MCSes) corresponding to the defect at hand.

The algorithm runs a loop that is invoked as long as flag `continue` is set to `''true''` (line 4). This condition is met while there are candidates that could become MCSes. The loop starts invoking the `getCandidateSubsets` function. This function is responsible for building the candidate subsets to become MCS; this is explained later with an example. This function receives three inputs: the first input is the variable `k`, which indicates the size of the subsets to be generated, the second input is the set of *constraints of the model* `(mc)`, from which the candidate subsets to become MCSes will be identified. Finally, the third input is the collection of MCSes identified in previous executions. These subsets are all subsets of size `k` (with the exception of the empty set), which can be formed with the constraints belonging to `(mc)` and that are not super sets of the MCSes previously identified.

For example, for the dead feature *ASP* (see Figure 1) the algorithm identifies a MCS of two elements (`k`=2). This MCS is formed by the constraints that represent the dependencies *D11* and *D14* (see Table 1). Assuming that the algorithm will identify MCSes of three elements (`k` = 3), the `getCandidateSubsets` function will return the subsets candidates that could become MCSes. These subsets of 3 elements shall not contain the constraints corresponding to dependencies *D11* and *D14* at the same time. In terms of our FM, this means that eliminating dependencies *D11* and *D14* fixes the dead feature *ASP*. Therefore, it would make no sense to evaluate whether eliminating these two dependencies and other ones will fix the defect because the correction would cease to be minimal.

Once all candidates to MCSs are found, the algorithm stores in the variable `subsets` the resulting collection of subsets, and evaluates each subset (i.e., `candidateMCS` ∈ `subsets`) as follows:

The algorithm creates an unsolvable constraint program α, formed by the union of the *constraints of the model* (`mc`), the *fixed constraints* (`fc`) and the *verification constrains* (`vc`) (line 8). Then, it removes from α the constraints of the candidate subset `candidateMCS` and evaluates if the resulting set of constraints α' is solvable (lines 9 and 10). Only if the α' set is solvable, the subset `candidateMCS` is a MCS. In that case, the subset `candidateMCS` is added to the list `MCSes` (line 11).

Once the algorithm terminates the evaluation of all candidate subsets to be MCSes of size `k`, it increments by one the value of `k` (line 14) and repeats the creation and evaluation of candidate subsets (lines 6-17). When the function `getCandidateSubsets` delivers an empty set, that means that there are no candidate subsets of size `k` to be MCSes. Then, the algorithm assigns `''false''` to the variable `''continue''` (line 16), returns the collection of identified MCSes (line 19) and terminates.

It is worth noting that the algorithm builds again the α (line 8) set for each candidate to be MCS (i.e., `candidateMCS`) still available. This way, the identification of each MCS always starts with the same unsolvable constraints program. Therefore, all identified MCSes are independent of each other.

In the case of This way, the identification of each MCS always starts with the same unsolvable constraints program. Therefore, all identified MCSes are independent of each other.dead feature *ASP*, Algorithm 1 identifies MCSes with one, two, and three constraints (see Table 1). This means that there are not MCSes with more than 3 constraints being minimal for dead feature *ASP*.

Once Algorithm 1 identifies the MCSes of each defect, these MCSes are expressed as constraints, which is not easy to understand for end users. The idea in our method is that designers would be able to understand the corrections identified by the method, even without knowing about constraint programming. For this reason, once the method identifies all MCSes for a defect, it searches the dependencies of the FM corresponding to each constraint of the MCSes. Then, the method searches, for each constraint of the MCS in the *Transformed model*, its correspondence in natural language. Finally, the method replaces each MCS by its corresponding natural language expression.

Table 1 shows the MCSes identified in this third step for*ASP* dead feature (see Figure 1). Thanks to the natural language description of each MCS, it is possible to identify which dependencies should be eliminated from the corresponding model to correct the defect at hand (in our case, the dead feature *ASP*).

Each MCS allows correcting the corresponding defect. For example, the first MCS in Table 1 indicates that the exclusion dependency between features *HTTPS* and *Milliseconds* (cf. *D22*) should be eliminated. This correction makes sense since that exclusion dependency prevents *HTTPS* feature to be selected in at least one product derived from our FM running example. *HTTPS* it is required by the *Dynamic* feature (cf. *D27*) and it is therefore necessary if we select the feature *ASP*. Thereby, by eliminating the dependency suggested by the first MCS, the contradiction between the dependency *D22* and *D27* are solved and the *ASP* feature will not be dead.

Table 1
Types of dependencies in Feature Models

Id	Correction	IdDepen	#Elem
MCS1	Traversal dependency between HTTPS and Miliseconds	D22	1
MCS2	Traversal dependency between Static and Flash	D24	1
MCS3	Traversal dependency between Reports and Miliseconds	D21	1
MCS4	Traversal dependency between Dynamic and HTTPS	D27	1
MCS5	Mandatory dependency between Additional services and Reports	D7	1
MCS6	Mandatory dependency between Additional services and Banners	D8	1
MCS7	Optional dependency between Banners and Flash	D10	1
MCS8	Optional dependency between Content and Static	D15	1
MCS9	Group cardinality between Dynamic and [ASP, PHP, JSP]	D18	1
MCS10	Mandatory dependency between Webserver and Content, Optional dependency between Content and Dynamic	D11,D14	2
MCS11	Mandatory dependency between Website and Webserver, Optional dependency between WebServer and Protocols, Optional dependency between Content and Dynamic	D3,D12,D14	3
MCS12	Mandatory dependenct between Website and WebServer, Group cardinality between Protocols and [FTP, HTTPS], Optional dependency between Content and Dynamic	D3,D16,D14	3

Algorithm 1 proposed algorithm to identify MCSes

Require: mc:constraints of the model, fc:fixed constraints, vc:verification constrains
Ensure: $MCSes$: MCSes collection

```
1:  k ← 1 // Size of subsets to analyze
2:  MCSes ← ∅
3:  continue ← true
4:  while (continue == true) do
5:    subsets ← getCandidateSubsets(k, mc, MCSes)
6:    if subsets ≠ ∅ then
7:      for all candidateMCS ∈ subsets do
8:        α ← mc ∪ fc ∪ vc
9:        α' ← α\{candidateMCS}
10:       if α' is solvable then
11:         MCSes ← MCSes ∪ {candidateMCS}
12:       end if
13:     end for
14:     k ← k + 1
15:   else
16:     continue ← false
17:   end if
18: end while
19: return MCSes
```

3.3.4 Analysis of identified corrections

Our method identifies 12 corrections for the dead feature *ASP* in the third step. This range of possibilities raises questions such as: which of all corrections is better? Is the preferable correction, the one involving the least number of changes to the model? Is it better to have different correction alternatives? We believe that designers of FMs are those who can answer these questions, because they are those who know the domain represented by a FM. That is why our approach, unlike others proposed in the literature [39, 40, 45], presents all MCSes that can be identified by systematically eliminating dependencies from a given FM. In consequence, designers can decide which correction to use according to the possible MCSes and their interests. These decisions would be, for instance, choosing the MCS that involves the minimum number of changes, or the MCS that involves keeping in the model some particular features or dependencies.

4 Implementation

All steps of the proposed method are systematic; therefore, they could be implemented in a computational tool. In particular, the three steps presented in Section 3 were implemented in a Java tool. The functionality of this tool will be integrated, in a near future, to *VariaMos* [20], our suite for variability models.

The tool that implements our method uses Java libraries to execute constraint programs in GNU-Prolog [7] or SWI-Prolog [46] in a way to guarantee termination and exhaustive search. Analyzed FMs should be expressed in the *SXFM* format (Simple XML Feature Model). The results of the analysis are exported into a XLS file. This file contains, in the first sheet, the defects identified for the FM at hand, and in the second sheet, the possible corrections identified for each defect. The tool and its installation manual are available on Internet [5] .

5 Preliminary Evaluation

The method proposed in this paper was evaluated preliminarily using 78 models with different features, with up to 120 dependencies. The models were analyzed with the tool that implements the proposed method. Preliminary evaluation focused on two aspects: accuracy and performance. The following subsections presents details of developed experiments and discusses the results.

5.1 Accuracy

Accuracy measures the degree of correctness of the results obtained from the method, compared to the expected values under two criteria: false positives and successes [11].

- **Falses positives:** correspond to correction subsets that were identified by the algorithm are not actual corrections. A correction is a false positive if: (i) the defect for which the correction was identified does not disappear. (ii) it is applied and it is not minimal.
- **Successes:** successes corresponds to the collection of effective minimal correction subsets identified by the proposed method. Particularly, a correction is a success if it is a minimal subset of dependencies that fixes at least one defect in the FM when they are removed from the model.

In order to detect false positives, we manually inspect the obtained results. This inspection was performed as follows for each correction identified by our method.

(i) An unsolvable constraint program representing the FM with the defect identified by the correction was created. Then, the restrictions that were part of each identified correction were manually removed from that constraint program.

(ii) We ran the resulting constraint program. If the constraint program becomes solvable, then the identified correction fixes the corresponding defect.

(iii) We assessed whether the correction is minimal. According to the MCSes definition (see Subsection 2.5). If we only remove a portion of a MCS to an unsolvable subset , the subset should remain unsolvable. Indeed, if the MCS is minimal, all their elements must be removed from α to turn into a solvable problem. In our manual inspection, we followed the same procedure, that is,

[5] https://github.com/lufe089/CLEI2014/tree/master/Herramienta

we removed only a part of the correction from the constraint program. If the constraint program remain unsolvable, then the correction is minimal.

When a correction fixes at least one defect and is minimal,it is a success. Otherwise, it is a false positive.

Three levels were defined to assess the accuracy of our method according to the criteria presented above. Those levels are: controlled, semi-controlled and randomized. All evaluated FMs used during this evaluation are available on Internet [6] .

(i) **Controlled**: at this level, we empirically evaluate the accuracy of our method usign two FMs available on scientific literature. The collection of possible corrections of this two FMs were alredy known. One of the FMs represents a home integration system [40], while the other represents a simplified version of the Ubuntu Operating System [8]. The accuracy at this level was evaluated by comparing the corrections obtained with our method and the results of the comparison models, resulting in a complete match. Consequently, our method can be considered accurate. In addition, we manually inspected the results and we found that were successes.

(ii) **Semi-controlled**: at this level, we evaluated the accuracy of the method using a case study where defects and some corrections were known (see Section 2.3). Defects were intentionally introduced into the model, so we knew in advance the expected corrections. However, since the method identified more correction than expected, the results were manually inspected too.

Overall, the method detected 187 corrections out of which 20 were common to more than one defect. This result is interesting, because it indicates that eliminating certain subset of dependencies could resolve more than one defect at the same time. For example, eliminating the cross-dependency between *Static* and *Flash* features fixes 3 defects: (i) dead features such as *ASP, Database, Dynamic, HTTPS, JSP, Minutes, PHP and Seconds*, (ii) false optional features such as *File, Flash, FTP, milliseconds, Additional services and Logs*; and (iii) redundancies such as the cross-tree dependencies between *File* and *FTP*, and between *Content* and *Protocols*.

(iii) **Randomized**: at this level, we evaluated the accuracy of the method on 25 FMs for which we did not know the corrections beforehand. On the one hand side, three of the 25 FM were proposed in product lines literature and are available in the SPLOT repository [23]. On the other hand, the remaining 22 FM were created with BeTTy [36] an automatic FM generator. In order to facilitate the manual inspection of the results, all FMs had less than 35 dependencies. In addition, each model had at least one of the defects presented in Section 3.2, and we consider at least one model for each type of defect.

It is worth noting that the method detected actual defective FMs, not defects that were intentionally introduced by us. This allows to handle bias created by false patterns in defect corrections that would have been reproduced by us under the form of defects in the sample. In addition, the sample models include

[6] https://github.com/lufe089/CLEI2014/tree/master/Modelos%20probados

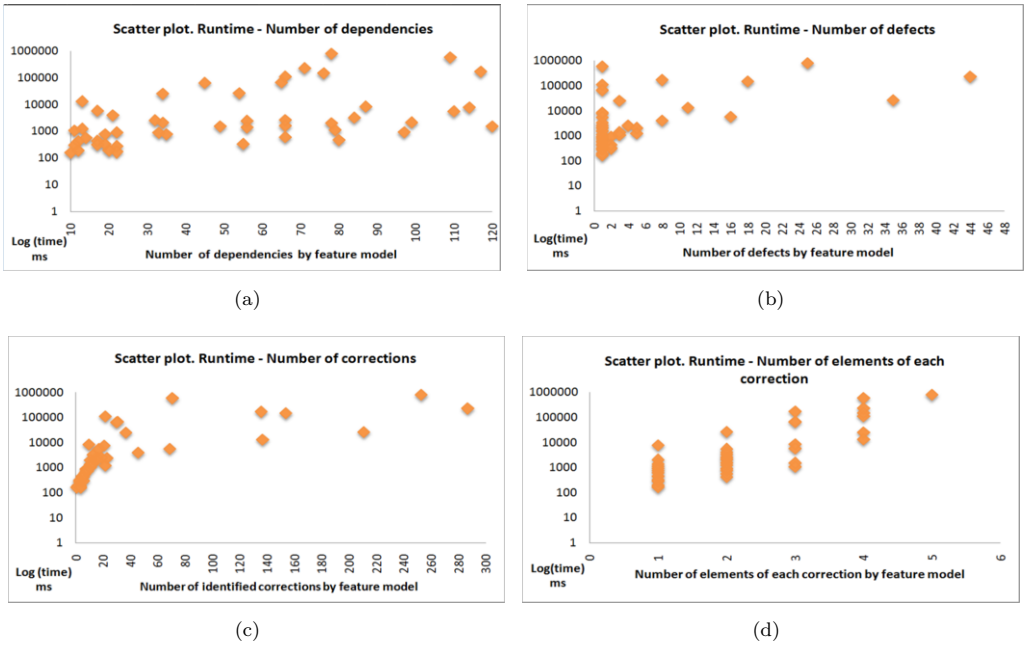

Figure 2. Scatter plot. Time vs [# dependencies, # defects, #corrections and # of elements of each correction]

different amount of defects, features and dependencies. These differentiating criteria were taken into account to measure the influence by defect patterns, the number of dependencies of FMs, the defect type, or by the number of defects.

Our results indicate that the proposed method is accurate because it found a 100 % success and 0% false positives. For sake of space, we do not detail here the analyzed models. Detailed results are available online [7].

5.2 Performance

An empirical evaluation of performance was carried out over 50 FMs automatically generated using BeTTy [36]. Next, we ran the tool that implements our method to identify corrections in these 50 models. Then, we measured the time consumed by the tool to perform this task. From this information, we compared the execution time considering four variables: number of dependencies, number of defects, number of corrections, and maximum number of elements in the identified corrections (see Figure 2). These variables were selected because of their influence on execution time. Tests were performed on a laptop running Windows 7 Ultimate 32-bit, processor Intel Core i5-2410M, 4.00 GB RAM, with 2.66 GB dedicated to the operating system. Each model has at least one of the defects presented in Subsection 2.2 and there is at least one model for each type of defect. The smallest model has 10 dependencies and the largest has 120 dependencies.

[7] https://github.com/lufe089/CLEI2014/tree/master/Resultados

To facilitate the analysis of results, Figures 2(a), 2(b), 2(c) and 2(d) shows in the y axis the runtime in a base 10 logarithmic scale, due to the large differences on times obtained analyzing the 50 considered models.

As Figure 2(a) shows, runtime is apparently not influenced by the number of dependencies of FMs. For example, the method required less time to identify defects and corrections in a FM with 120 dependencies than from FM with 80 dependencies. The reason might be that if a FM with 120 dependencies has less defects than a FM with 80 dependencies, the method might be able to analyze faster the FM with fewer defects. Figure 2(b) shows that runtime is apparently related to the number of defects. This indicates that the number of defects increases the execution time required to analyze models. Nevertheless, there were cases where our tool required more computation time analyzing FM with only one defect.

The runtime and the amount of corrections identified are apparently related. In fact, as Figure 2(c) shows, in some FM the runtime increases as long as the number of corrections growth. This means that it is expected that the method spend more time analyzing FMs with 200 corrections, than models with 20 corrections. Finally, it seems that the execution time is related to the maximum number of elements of each correction (see Figure 2(d)). Indeed, the execution time increases when the number of elements of each correction increases. Then, it is expected that the method takes more time analyzing a model whose corrections have four elements compared to the time required to evaluate a model which corrections contain a single element. This relationship makes sense, since the method does not end until it identify all minimum corrections in the FMs, no matter how many elements have each correction.

Based on the above, our preliminary performance evaluation indicates that it is difficult to predict how long our method takes to identify defects and propose corrections. According to the scattered plots, the runtime of the method is apparently related to the number of defects, number of corrections and size of corrections. Nevertheless, this information is only known once the FM is analyzed. Before analyzing a FM with the method proposed here, the model designer only knows the number of dependencies of the model. However, according to scattered plot in Figure 2(a), it seems that the runtime does not dependend on the number of dependencies of the analyzed model. Therefore, FMs with different number of dependencies might have the same execution time, while FMs with the same number of dependencies might have different execution times.

6 Related works

The scientific literature provides several approaches to automate the identification of semantic defects in FMs [4, 26, 35, 43, 45, 47]. Nevertheless, none of them is able to automatically explain how to correct the identified defects. The originality of our proposal is that it identifies not only the defects to be fixed, but also how they could be repaired.

Trinidad et *et al.* [40] transform FMs into a diagnostic problem. Then they

solve a constraint satisfaction program to identify the smaller size corrections for each defect identified. In this approach, the defects addresed are: void models, dead features, and false optional features. This proposal was automatized in FaMa [41], a framework developed in Java for automatic analysis of FMs. Nevertheless, this work does not identify other subsets of dependencies (not necessarily the smallest), that could be also eliminated to fix some defects. For instance, by considering the dead feature *ASP*, our proposal identifies corrections with one, two an three elements, whereas the proposal of Trinidad et *et al.* [40] only identifies corrections with one single element.

In a more recent work, Trinidad and Ruiz-Cortés [42] use abductive reasoning to explain why dead features, false optional and void models appears. Unfortunately, authors do not provide any details or algorithm to implement their proposal.

Wang et *et al.* [44] identify corrections for void FMs. In their work, the designer assigns priorities to the FM dependencies. Then, the authors use a tool to identify the minimum subset of dependencies with lower priority to be removed from FM, in order to obtain at least one product from the FM. This method only identifies one correction at a time, and it must be re applied if the proposed solution does not meet the user's interests. Furthermore, this work does not identify corrections for dead features, false optional features or redundancies, as it does our approach.

Noorian et *et al.* [25] propose a framework based on descriptive logic [2] to identify if any FM is void and to suggest possible corrections. The framework automatically transforms FMs expressed in *SXFM* format into an OWL-DL file. Then, the framework uses the ontological reasoner Pellet [37] to verify if the resulting OWL-DL file is inconsistent. In that case, the framework invokes a Pellets functionality to extract the minimal subsets of OWL axioms that must be removed from the OWL file to turn it into consistent. This proposal identifies corrections of void models and invalid configurations. However, it does not consider other defects such as dead features, false optional features, redundancies and false product lines, whereas our approach also deals with those defects. In addition, the framework presents corrections in OWL-DL pure, which is difficult to understand for the designer of FMs

Thüm *et al.* propose Feature IDE [39] a tool that supports the feature oriented software development and allows identifying corrections for void models, false optional features, and dead features. Particularly, Feature IDE automatically identifies what are the requires and excludes dependencies to be removed in order to fix each identified defect. However, Feature IDE only identifies corrections that implies removing one dependency from the FM. If correcting a defect implies to delete more than one dependency, then FeatureIDE does not identify any correction for that defect. Furthermore, FeatureIDE does not identify redundancies, false product line models, or their corrections.

Rincón *et al.* [31] use ontologies and a set of rules represented in an ontology query language, in order to (i) represent FMs, (ii) identify dead features and false optional features, (iii) identify some causes of these defects, and (iv) create an explanation in natural language. However, this proposal only identifies some defects

and causes. If the model has defects that do not correspond to any defined rules, then these defects will not be detected. This work focuses on identifying some causes of defective FMs, while the method proposed in this paper deals with identifying their corrections. Thus, both proposals are complementary and we could integrate them in order to identify defects, causes and corrections.

Regarding MCSes, Reiter [30] was one of the first authors to use MCSes to identify how to fix unsolvable constraint programs. Later on, other studies have proposed algorithms to identify MCSes in boolean constraint programs [3, 14–16], and integer constraint programs [9, 17]. However, these approaches are focused on proposing algorithms to identify MCSes, while our proposal is focused on applying the concept of MCSes in defective FMs, which can be represented as constraint programs. One of our future works, is oriented to testing and compare the algorithms proposed in literature, in order to optimize our identification of MCSes.

In a previous work, Rincón *et al.* [32] propose a semi-automatic method to explain why each dead feature occurs in FMs. This approach implies to manually transform FMs into a constraint satisfaction problem, and then they identify all the minimal corrections subsets of dependencies that could be modified to correct each dead feature of the FM. This approach like FaMa [41], identify the list of dependencies that entail the fewest changes to fix the defect, but also identify others set of dependencies that imply more changes and fix the defect. This information provides more complete information about how correcting each dead feature. The method presented in this paper is a continuation of that research. For this reason, we also identify the MCSes, but with a fully automated method. Furthermore, our current method deals with more types of defects (see Subsection 3.3.2), and delivers corrections in natural language to facilitate the understanding of possible corrections of defects of FMs.

7 Conclusions and future works

High quality FMs are essential to take full advantage of the benefits provided by product lines. In this paper, we presented a novel method that allows no only identifying semantic defects in FMs, but also the corrections for each defect.

To operationalize our proposal, we identify minimal corrections subsets (MCSes). In our specific case, these are the minimal subset of dependencies that should be removed from a FM to correct at least one of their defects. MCSes are identified using constraint programs. However, our method automatically traduces the identified MCSes into natural language, in order to provide understandable results. In this way, we use constraint programming as an intermediate language for reasoning on FM, but this is completely transparent for the FM designer. As indicated in [12, 13, 27, 38, 40], all information supporting the correction process help to save time and cost in the development of the product line. Therefore, we believe that to know the corrections of each defect as soon as possible, would allow designers to focus on creating good representations of the product line domain, rather than finding how to correct FMs defects.

As far as we know our approach is the first proposal that identifies all corrections that can be detected by systematically eliminating dependencies from a given FM. This provides in our opinion more complete information about how to correct each defect, because designers might decide according to their interests which correction to use. However, there are other cases out of the scope of this proposal. For example, our method only identifies corrections that involve removing dependencies. Other corrections that require updating the model, extending the variables domain or adding new dependencies are not yet considered. Additionally, we believe that it might be useful to propose some criteria to make the best correction selection easier for the designer. Some possible criteria would be: if a correction generates new defects, if it restricts the amount of products derivable from FMs or if the correction repairs at the same time more than one defect in the FM. Furthermore, our future line of research is interested on extending the proposed method to identify for each defect not only their corrections but also their causes.

Acknowledgments

The research presented in this paper is part of Project No. RC 0634-2013, titled "Development of solutions to support the completeness and correctness of product lines with application to software engineering." This project is funded by the Administrative Department of Science, Technology and Innovation (COLCIENCIAS) of the Republic of Colombia.

References

[1] Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Reveal: A Formal Verification Tool for Verilog. In Andrei Cervesato, Iliano and Veith, Helmut and Voronkov, editor, *Logic for Programming Artificial Intelligence and Reasoning (LPAR-2008)*, volume 5330, pages 343–352. 2008.

[2] Franz Baader, Diego Calvanese, Deborah L McGuinness, Daniele Nardi, and Peter F Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications.* Cambridge University Press, New York, NY, USA, 2003.

[3] James Bailey and Peter J Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05). Lecture Notes in Computer Science*, pages 174–186, Long Beach,USA, 2005. Springer Berlin Heidelberg.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines SPLC'05*, SPLC'05, pages 7–20, Rennes, France, 2005. Springer-Verlag.

[5] Paul Clements and Linda M Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 1st edition, 2001.

[6] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[7] Daniel Diaz and Philippe Codognet. The GNU prolog system and its implementation. *Design and Implementation of the GNU Prolog System*, 2001.

[8] Alexander Felfernig, David Benavides, J Galindo, and F Reinfrank. Towards Anomaly Explanation in Feature Models. In *Proceedings of the Workshop on Configuration*, number 827587, pages 117–124, Vienna, Austria, 2013.

[9] Alexander Felfernig, M Schubert, and C Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26(1):53–62, 2012.

[10] Marco Gavanelli and Francesca Rossi. Constraint Logic Programming. In *A 25-year perspective on logic programming*, volume 6125, pages 64–86. Springer-Verlag Berlin, January 2010.

[11] ISO-5725. International Standard ISO 5725-1, Accuracy (trueness and precision) of measurement methods and results General Principles and definitions, 1994.

[12] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and Spencer Peterson Peterson. Feasibility Study Feature-Oriented Domain Analysis (FODA). Technical Report. Technical report, Software Engineering Institute,Carnegie Mellon University, 1990.

[13] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality Assurance in the Presence of Variability. In Selmin Nurcan, Camille Salinesi, Carine Souveyet, and Jolita Ralyté, editors, *Intentional Perspectives on Information Systems Engineering*, pages 319–333. Springer Berlin Heidelberg, 2010.

[14] Mark H Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer Berlin Heidelberg, 2013.

[15] Mark H Liffiton and Karem Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[16] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, and Anton Belov. On Computing Minimal Correction Subsets. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, Beijing, China, 2013.

[17] Joao Marques-Silva and Inês Lynce. On improving MUS extraction algorithms. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 159–173, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Raúl Mazo. *A Generic Approach for Automated Verification of Product Line Models*. Ph.d.thesis, Ph.D.dissertation.Paris 1 Panthéon Sorbonne University, Paris, France, 2011.

[19] Raúl Mazo, Camille Salinesi, and Daniel Diaz. Abstract Constraints: A General Framework for Solver-Independent Reasoning on Product-Line Models. *Journal of International Council on Systems Engineering (INCOSE)*, 14(4):22–24, 2011.

[20] Raúl Mazo, Camille Salinesi, and Daniel Diaz. Variamos: a tool for product line driven systems engineering with a constraint based approach. In *CAiSE Forum*, pages 147–154, 2012.

[21] Raúl Mazo, Camille Salinesi, Daniel Diaz, Olfa Djebbi, and Alberto Michiels. Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. *International Journal of Information System Modeling and Design IJISMD*, 3(2):33–68, 2011.

[22] Marcilio Mendonça, Thiago Tonelli Bartolomei, and Donald Cowan. Decision-making coordination in collaborative product configuration. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 108–113, New York, NY, USA, 2008. ACM.

[23] Marcilio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[24] Antonio Morgado, Mark H Liffiton, and Joao Marques-silva. MaxSAT-Based MCS Enumeration. In *Haifa Verification Conference (HVC 2012)*. Springer, 2012.

[25] Mahdi Noorian, Alireza Ensan, Ebrahim Bagheri, Harold Boley, and Yevgen Biletskiy. Feature Model Debugging based on Description Logic Reasoning. In *DMS'11*, pages 158–164, 2011.

[26] Abdelrahman Osman, S Phon-Amnuaisuk, and Chin Kuan Ho. Knowledge Based Method to Validate Feature Models. In *First International Workshop on Analyses of Software Product Lines*, pages 217–225, 2008.

[27] Abdelrahman Osman, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Investigating Inconsistency Detection as a Validation Operation in Software Product Line. In Naohiro Lee, Roger and Ishii, editor, *Software Engineering Research, Management and Applications*, volume 253, pages 159–168. Springer Berlin Heidelberg, 2009.

[28] Barry O'Sullivan, Alexandre Papadopoulos, Boi Faltings, and Pearl Pu. Representative explanations for over-constrained problems. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, AAAI'07, pages 323–328. AAAI Press, 2007.

[29] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[30] R Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[31] Luisa Rincón, Gloria Lucia Giraldo, Raúl Mazo, and Camille Salinesi. An ontological rule-based approach for analyzing dead and false optional features in feature models. In *XXXIX Latin American Computing Conference (CLEI)*, 2013.

[32] Luisa Fernanda Rincón Perez, Gloria Lucía Giraldo Gómez, Raúl Mazo, Camille Salinesi, and Daniel Diaz. Subconjuntos Minimos de Correccion para explicar caracteristicas muertas en Modelos de Lineas de Productos. El caso de los Modelos de Caracteristicas. In *Proceedings of the 8th Colombian Computer Conference (CCC)*, Armenia-Colombia, 2013.

[33] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, volume 35. Elsevier Science Inc, 2006.

[34] Sean Safarpour, Hratch Mangassarian, Andreas Veneris, Mark H Liffiton, and Karem A Sakallah. Improved Design Debugging Using Maximum Satisfiability. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, pages 13–19, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Camille Salinesi and Raúl Mazo. Defects in Product Line Models and how to identify them. In Abdelrahman Elfaki, editor, *Software Product Line - Advanced Topic*, chapter 5, pages 97–122. InTech, 2012.

[36] Sergio Segura, José A Galindo, David Benavides, José A Parejo, and Antonio Ruiz-Cortés. BeTTy: benchmarking and testing on the automated analysis of feature models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 63–71, New York, NY, USA, 2012. ACM.

[37] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[38] Jing Sun, Hongyu Zhang, and Hai Wang. Formal Semantics and Verification for Feature Modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '05, pages 303–312, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Thomas Thüm, Christian Kastner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[40] Pablo Trinidad, David Benavides, A Duran, Antonio Ruiz-Cortés, and M Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[41] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. FAMA Framework. In *Proceedings of the 12th International Software Product Line Conference*, SPLC '08, page 359, Washington, DC, USA, 2008. IEEE Computer Society.

[42] Pablo Trinidad and Antonio Ruiz-Cortés. Abductive Reasoning and Automated Analysis of Feature models: How are they connected. In *Proceedings of the Third International Workshop on Variability Modelling of Software-Intensive Systems*, pages 145–153, 2009.

[43] T Von der Massen and H Lichter. Deficiencies in Feature Models. In Tomi Mannisto and Jan Bosch, editors, *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[44] B Wang, Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Wei Zhang, and Hong Mei. A dynamic-priority based approach to fixing inconsistent feature models. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 181–195, Berlin, Heidelberg, 2010. Springer-Verlag.

[45] Hai Wang, Yuan Li, Jing Sun, Hongyu Zhang, and Jeff Pan. Verifying feature models using OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):117–129, June 2007.

[46] Jan Wielemaker. SWI Prolog Reference Manual (Version 6.2.2), 2012.

[47] Wei Zhang and Haiyan Zhao. A Propositional Logic-Based Method for Verification of Feature Models. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, pages 115–130, Seattle-USA, 2004.