Automated Error Analysis for the Agilization of Feature Modeling [1]

P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, M. Toro

Dpto. de Lenguajes y Sistemas Informáticos, University of Seville
Av. Reina Mercedes s/n, 41012 Seville (Spain)

Abstract

Software product lines (SPL) and agile methods share the common goal of rapidly developing high quality software. Although they follow different approaches to achieve it, some synergies can be found between them by *i*) applying agile techniques to SPL activities so SPL development becomes more agile; and *ii*) tailoring agile methodologies to support the development of SPL. Both options require an intensive use of feature models, which are usually strongly affected by changes on requirements. Changing large–scale feature models as a consequence of changes on requirements is a well–known error–prone activity. Since one of the objectives of agile methods is a rapid response to changes in requirements, it is essential an automated error analysis support in order to make SPL development more agile and to produce error–free feature models.

As a contribution to find the intended synergies, this article sets the basis to provide an automated support to feature model error analysis by means of a framework which is organized in three levels: a feature model level, where the problem of error treatment is described; a diagnosis level, where an abstract solution that relies on Reiter's theory of diagnosis is proposed; and an implementation level, where the abstract solution is implemented by using constraint satisfaction problems (CSP).

To show an application of our proposal, a real case study is presented where the Feature–Driven Development (FDD) methodology is adapted to develop an SPL. Current proposals on error analysis are also studied and a comparison among them and our proposal is provided. Lastly, the support of new kinds of errors and different implementation levels for the proposed framework are proposed as the focus of our future work.

Keywords: feature models, agile methods, error analysis, theory of diagnosis, constraint programming

1 Introduction and Motivation

The so–called *agile methods* have arisen to face up to the problems that traditional, *heavyweight* software development methodologies have not satisfactorily solved yet. Agile methods pursue some main goals which are described in the *Agile Manifesto* (Fowler and Highsmith, 2001), where a number of key changes to traditional software development are proposed. For example: focusing the efforts during development in the interaction with customers through working software; collaborating with customers during development instead of negotiating contracts at the beginning of development; adapting software to changing requirements, etc. In other words, the aim of agile methods is producing high–quality software products in less time and cost than using traditional software development methodologies by reducing unnecessary tasks and increasing productivity.

On the other hand, the *software product line* (SPL) approach intend to develop a set or family of software products within a concrete application domain. In a SPL, software products are developed from a set of shared, common assets —the *core assets*— and a set of *product–specific assets*. The core–assets development process is known as *domain engineering* whereas the product–specific assets development process is known as *application engineering* (Pohl et al., 2005).

Although both approaches are very different from each other, they share the aim of reducing development time and cost while quality is not compromised, even increased. Our experience applying both approaches separately has made us think that is possible to find some synergies by sharing some practices and techniques so that SPL development becomes more *agile* and agile methods can adopt an SPL–like orientation.

Considering the *agilization* of SPL development, this article is focused on processes related to the so–called *feature models*, which are used to describe the products in an SPL and are intensively used in SPL development (see Section 2.1). For example, Czarnecki et al. (2005) and Sochos et al. (2004) propose inferring the core architecture from feature models; Batory et al. (2004) propose using *feature–oriented programming* (FOP) to implement an SPL decomposing the architecture into features and automatically deriving products from a selection of their features; Benavides et al. (2005) use feature models to support decision making during production.

Email address: {ptrinidad, benavides, amador, aruiz, mtoro}@us.es (P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, M. Toro).

[1] This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web–Factories (TIN2006–00472).

As many processes in SPL development use feature models, applying some agile principles to frequent operations on feature models can make SPL development more *agile*, especially during the domain engineering process in which large–scale feature models must be developed. As Kang et al. (1990) stated in the *Feature–Oriented Domain Analysis* (FODA) report, an important task when using large–scale feature models efficiently is checking that they contain no errors after the introduction of changes, something that cannot be performed manually. Despite of the need of automatic support for error analysis in feature models, there is a lack of proposals that focus on producing error–free feature models (see Section 7). Taking all this into consideration, providing automated support for feature model error analysis can be considered as needed step toward the agilization of SPL development.

The other synergistic approached taken into consideration is tailoring an agile methodology to introduce SPL orientation. For that purpose, the agile methodology that fits better with the principles of SPL has been chosen. Williams (2004) compares 4 agile methodologies (XP, FDD, SCRUM and Crystal) and concludes that FDD is the agile methodology that has the most thorough analysis and design practices. FDD decomposes the customer requirements in terms of features to obtain a list of features. An iterative and incremental process is defined to develop one or more features from the features list and constantly deliver the software to the customer in two–weeks iterations. After each iteration and depending on the customer's feedback, the list of features is reviewed. The proposed SPL orientation of FDD is based on three common points:

- SPL and FDD decomposes the software in terms of features.
- The list of features and feature models are evolutionary as they are constantly reviewed.
- FDD invest an important effort in analysis and design and agility is introduced in the two–weeks iterations.

Introducing feature models in FDD to support SPL principles also implies that whenever a feature model changes in an iteration, it must be checked to be error–free. As mentioned before, this is an error–prone activity especially when dealing with large–scale SPLs. Therefore, an automated support for feature modeling is also needed to adapt FDD to SPL principles.

In conclusion, a first step in combining SPL and agile methods, independently from the alternative chosen, implies the need of an automated technique supporting the production of error–free feature models.

Despite of the demand of the FODA report for an automated support to produce error–free feature models and years after it was published, this demand still remains and has only been partially dealt with. Previous works such

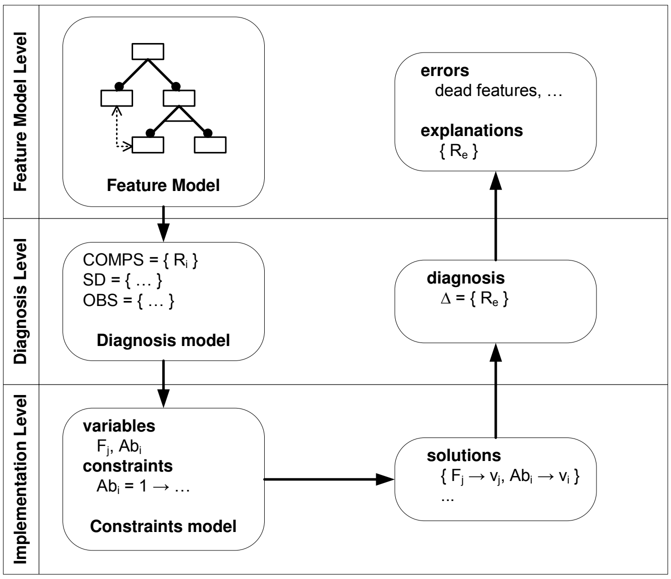


Figure 1. Feature model error treatment framework

as (Mannion, 2002; von der Massen and Lichter, 2004; Czarnecki and Kim, 2005; Raatikainen et al., 2005) have partially dealt with detecting errors in feature models. Apart from Batory (2005), few of them have focused on the importance of producing error–free feature models, including the possibility of providing explanations to the modeler so that errors can be detected and removed in an agile way. The work presented in this article contributes to automating the error treatment in feature models by modeling the problems of detecting and explaining errors and providing operational solutions to both of them. In this way, we think feature modeling can be agilized, a step towards SPL agilizing.

This article proposes automating error treatment of feature models using a *three–level framework*, which is depicted in Figure 1. The *feature model level*, which describes the problem of detecting and explaining errors in feature models, is presented in Section 3. In Section 4, the *diagnosis level*, which maps feature models onto *diagnosis models* using *theory of diagnosis* to formally describe the problem of detecting and explaining errors, is described. In Section 5, the *implementation level*, which implements the diagnosis level using constraint satisfaction problems, a descriptive technique that can be solved using off–the–shelf solvers, is described. Additionally, Section 2 presents some preliminaries where feature models are briefly described and some basic concepts of theory of diagnosis and constraint satisfaction optimization problem are introduced. In Section 6 we validate our proposal by applying it to a real case study. Section 7 summarizes the related work on automated treatment of errors in feature models. Finally, Section 8 shows some conclusions and the paths to follow in our future work in error analysis and agile software product line.

2 Preliminaries

2.1 Feature Models

As briefly commented in Section 1, feature models are a widely used notation to describe the set of products in a software product line in terms of features. In feature models, features are hierarchically linked in a tree–like structure and are optionally connected by cross–tree constraints. An example on how feature models are depicted is shown in Figure 2, where the feature model describes a *Home Integration System*(HIS) product line.

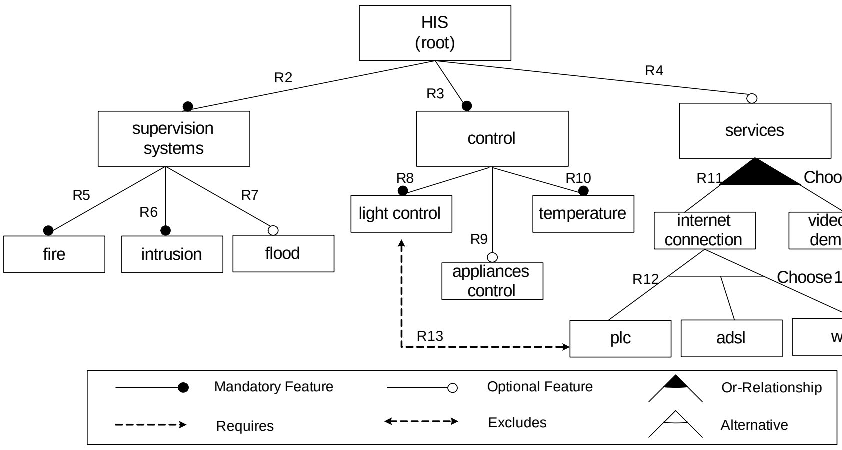


Figure 2. Home Integration System (HIS) feature model diagram

Although there are many proposals on the type of relationships and their graphical representation in feature models (see the work by Schobbens et al. (2007) for a detailed survey), the most usual relationships are the following:

Mandatory A *child feature* is *mandatory* when it is required to appear whenever its *parent feature* appears. In the example, `R10` is a mandatory relationship between `control` (the parent feature) and `temperature` (the child feature), i.e. whenever a temperature control is present in a product, there must be a control system in that product.

Optional A child feature is said to be *optional* when it can appear or not whenever its parent feature appears. In the example, `R9` is an optional relationship between `control` and `appliances control`, i.e. the appliances control feature can be optionally chosen whenever there is a control system in a product.

Or–relationship A set of child features have an *or–relationship* with their parent feature when one or more child features can be selected when the parent feature appears. Relationship `R11` in Figure 2 is an or–relationship where whenever `services` is selected, `video on demand` or `internet connection` or both must be selected.

Alternative A set of child features are said to be *alternative* when only one of them must be selected when their parent feature appears. Relationship `R12` is an alternative relation in Figure 2, where `adsl`, `plc` or `wifi` internet connections must be selected but only one of them in a single product.

Requires, Excludes A cross–tree relationship like `A requires B` means that in any product where feature `A` appears, feature `B` must also appear. On the other hand, a relationship like `A excludes B` means that both features cannot appear in the same product at the same time. In the sample feature model, `plc` cannot appear in a product if `light control` appears and vice versa.

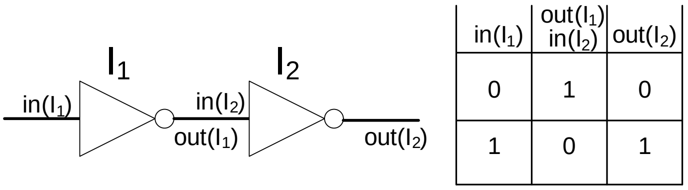
2.2 Theory of Diagnosis

The well–known *theory of diagnosis* proposed by Reiter (1987) has been widely used to *diagnose* systems—especially electronic circuits—, i.e. to determine which system components, if any, make the system behave abnormally.

In Reiter's theory of diagnosis, a system is modeled as a pair $(SD, COMPS)$ where $COMPS$ is the set of system components and the system description (SD) is a set of predicates defining the *behavioral* and *structural* models of the system. In the behavioral model, the *normal* behavior of system components is described as logical implications of the negation of their *abnormal behavior*, denoted as $Ab(c)$ where c is a system component. Obviously, the negation of an abnormal behavior is considered as a *normal behavior*.

As an example inspired by the one developed by de Kleer et al. (1990), let us consider the two–inverter circuit in Figure 3. An inverter is an digital electronic component that inverts its input, i.e. it outputs 1 when its input is 0 and vice versa. As the reader can imagine, the normal behavior of the circuit in Figure 3 is outputting its input since is a double inversion, i.e. $in(I_1) = out(I_2)$.

Following Reiter's theory of diagnosis, the system components can be modeled as $COMPS = \{\ I_1,\ I_2\ \}$, representing the two inverters. Assuming an inversion function $inv : \{0, 1\} \rightarrow \{0, 1\}$ such that $inv(0) = 1$ and $inv(1) = 0$, the behavioral and structural models, i.e. the *system description SD*, would be the following:



$in(I_1)$	$out(I_1)$ $in(I_2)$	$out(I_2)$
0	1	0
1	0	1

Figure 3. The two–inverters circuit and its normal behavior

$$\begin{aligned} SD = \{&\neg Ab(I_1) \Rightarrow out(I_1) = inv(in(I_1)), && \text{[behavioral model]}\\ &\neg Ab(I_2) \Rightarrow out(I_2) = inv(in(I_2)), && \text{[behavioral model]}\\ &out(I_1) = in(I_2)\ \} && \text{[structural model]} \end{aligned}$$

In Reiter's theory of diagnosis, in order to diagnose a system a set of observations OBS, expressed as predicates, is needed. For example, a set of observations for the two–inverter circuit —denoting an abnormal behavior— could be $\{\ in(I_1) = 0,\ \ out(I_2) = 1\ \}$. In this context, a Reiter's diagnosis of $(SD, COMPS, OBS)$ is defined as a minimal set of components with abnormal behavior, denoted as Δ. In other words, $\Delta\ \subseteq COMPS$ is a *diagnosis* of $(SD, COMPS, OBS)$ if the following set of predicates is consistent and Δ is minimal:

$$SD\ \cup\ OBS\ \cup\ \{\ Ab(c) \mid c \in \Delta\ \}\ \cup\ \{\ \neg Ab(c) \mid c \in COMPS - \Delta\ \} \quad (1)$$

If the system behaves normally, then $\Delta = \varnothing$ and the following set of predicates is consistent:

$$SD\ \cup\ OBS\ \cup\ \{\ \neg Ab(c) \mid c \in COMPS\ \} \quad (2)$$

Following with the two–inverters example and the former example observation, to check if the system behaves normally, i.e. if $\Delta = \varnothing$, the following set of predicates must be verified to be consistent:

$$\begin{aligned} \{\ &\neg Ab(I_1) \Rightarrow out(I_1) = inv(in(I_1)), && \text{[SD (behavioral model)]}\\ &\neg Ab(I_2) \Rightarrow out(I_2) = inv(in(I_2)), && \text{[SD (behavioral model)]}\\ &out(I_1) = in(I_2), && \text{[SD (structural model)]}\\ &in(I_1) = 0, out(I_2) = 1, && \text{[OBS]}\\ &\neg Ab(I_1), \neg Ab(I_2)\ \} && [\neg\text{Ab(c)} \mid \text{c} \in \text{COMPS}] \end{aligned}$$

In this case, the previous set is not consistent. That means that there are components in Δ that makes the system behave abnormally. A naive algorithm to identify the components with abnormal behavior is to check all elements in the power set of components $\mathbb{P}\,COMPS$ against formula (1) and select those which are minimal. In the example, $\mathbb{P}\,COMP\ =\ \{\ \varnothing, \{I_1\}, \{I_2\}, \{I_1, I_2\}\ \}$ and two diagnoses, $\Delta_1 = \{I_1\}$ and $\Delta_2 = \{I_2\}$ make formula (1) consistent and are minimal. That means that only one of the inverters fails, but not the two of them simultaneously.

There are many different techniques to diagnose a system based on Reiter's theory of diagnosis. In Section 5 we show how to use constraint satisfaction problem solvers to diagnose feature models, applying the concepts described in this section.

A *Constraint Satisfaction Problem (CSP)* is a declarative paradigm to model and solve problems using *constraints* (Tsang, 1995). A CSP is defined as a 3–tuple (V, D, C) where V is a set of variables, each ranging on a finite domain from set D, and C is a set of constraints restricting the values that the variables can take simultaneously. A solution to a CSP is an assignment to each variable of a value from its corresponding domain so that all constraints are satisfied simultaneously. In the common usage of CSPs, we may search for: *i*) just one solution, with no preference, *ii*) all solutions, *iii*) an *optimal solution* by means of an *objective function* defined in terms of one or more variables of the problem.

Consider for instance, the CSP: $(\{a, b\}, \{\ \{0, 1, 2\},\ \{0, 1, 2\}\ \}, \{a + b\ <\ 4\})$ where both variables a and b take value in the domain $\{0, 1, 2\}$ and are constrained by $\{a + b\ <\ 4\}$. The only value assignment that does not satisfy $a + b < 4$ is $\{a \mapsto 2, b \mapsto 2\}$, so there are eight solutions. Nevertheless, if we replace the constraint with $a + b < 0$ then the CSP is not *satisfiable*, i.e. there is no possible value assignment satisfying the constraints.

In many real–life applications, we do not want to find any solution to a CSP but a good one. The quality of a solution is usually measured by an application–dependent function called *objective function*. In these cases, the goal is finding a solution that satisfies all the constraints and minimize or maximize the objective function. Such problems are referred to as *Constraint Satisfaction Optimization Problems*, that consist of a CSP (V, D, C) and an optimization function O that maps every solution to a numerical value.

In the previous example, suppose that we define a constraint satisfaction optimization problem where the optimization function is $O(s) = a$, which maximizes the value of a. There are two solutions in the original CSP, $\{\ \{a \mapsto 2, b \mapsto 0\},\ \{a \mapsto 2, b \mapsto 1\}\ \}$, that maximizes the value of the objective function and are therefore the solutions of the constraint satisfaction optimization problem.

There is an important amount of research on algorithms and heuristics to solve constraint satisfaction (optimization) problems, and the set of operational alternatives is growing, including both commercial and free solvers.

3 Feature Model Level: Dealing with Errors in Feature Models

A feature model is composed by features and relationships among them. A feature model describes the products in a SPL, considering products as sets of selected features. Relationships are added to reduce the set of products until the SPL is properly described.

Sometimes, introducing new relationships in a feature model may accidentally remove some products so the feature model does not describe the real SPL. On the other hand, the feature model may not be correctly constrained so some products that are not in the SPL are still kept in the feature model. Therefore, feature modeling is an error–prone task where representing the correct SPL in terms of features and relationships is not as easy as it seems.

We consider that an error is an incorrect definition of relationships that suggests that the set of products described by a feature model may not match the SPL it describes. Although this definition could cover many kinds of errors, in this article we focus on three kinds that have already been considered in the bibliography:

Dead features A *dead feature* is a non–instantiable feature, i.e. a feature that despite of being defined in a feature model, it appears in no product in the software product line. Common cases where dead features are generated are shown in Figure 4.

Full–mandatory features A child feature in a non-mandatory relationship is a *full–mandatory* feature if it has to be instantiated whenever its parent feature is, i.e. it is neither an optional nor an alternative feature. The most common cases are shown in Figure 5. Full–mandatory features usually appear together with dead features, as can be observed for some cases in Figures 4 and 5.

Void feature models A feature model is *void* if it defines no product at all.

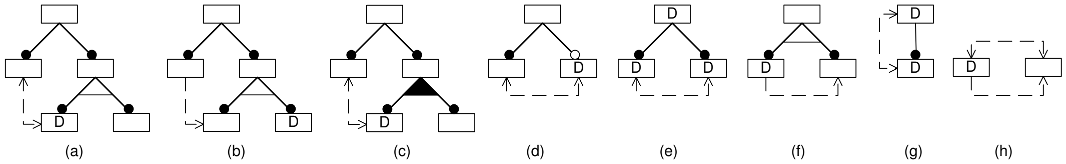


Figure 4. Common cases of dead features

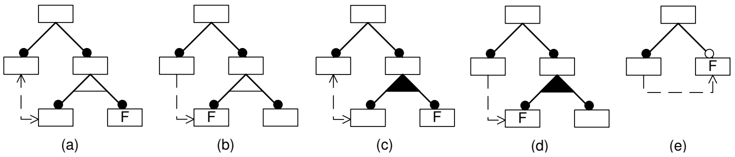


Figure 5. Common cases of full-mandatory features

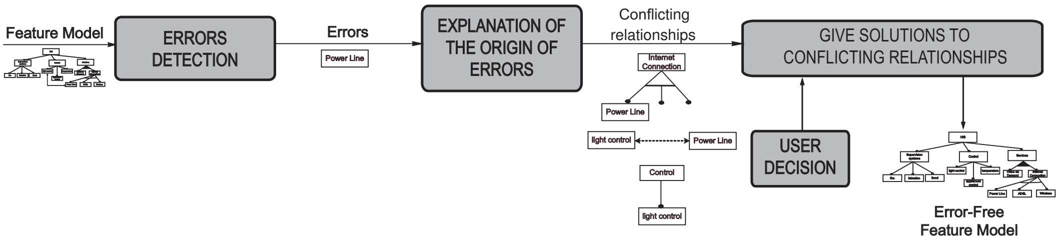


Figure 6. Two steps process to analyze errors

This error is commonly caused by contradictory relationships among mandatory features. Void feature models are also known as *inconsistent*, *invalid* or *unsatisfiable* feature models (Batory et al., 2006) although this expression is closely related to the method used to detect them in the literature. As a void feature model defines no product, no feature is instantiable. That means that every feature is dead including the root. So we can conclude that a void feature model is the one whose root is a dead feature.

Some authors (see Section 7) have detected that the main complexity of producing error–free feature models relies on modifying the right relationships to remove the errors. If no explanation of the source of errors is provided, the production of error–free feature models relies on the skills of the feature modeler. Our objective is assisting the feature modeler in making decisions to produce error–free feature models. We propose the following two steps to provide the feature modeler not only the list of errors within a feature model but also the explanations for the relationships that cause the errors, as shown in Figure 6:

(1) **Detection**: This step focuses on detecting the features that are affected by the errors from a given feature model that is received as an input. This step searches for dead and full-mandatory features (a void feature model is a particular case of dead feature) and outputs a list of them.
(2) **Explanation**: For each feature in the list provided by the previous step, all the explanations that are the origin of the errors are provided. An explanation consists of one or more relationships that must be modified to remove an error. For each error, many explanations can be given. Using case (d) in Figure 4 as an example, the `excludes` relationship is an explanation for the dead feature because it can be removed to solve the error. The mandatory relationship can also be transformed into an optional relationship to solve the error, so this relationship is another explanation for the dead feature.

Following this process, the feature modeler can use the explanations provided to correct the errors and producing an error–free feature model. This article describes an automated support for the detection and explanation of errors in feature models as a contribution toward agile feature modeling.

4 Diagnosis Level: Diagnosing and Explaining Errors

The goal of the diagnosis level, as depicted in Figure 1, is transforming a feature model into a diagnosis model in order to detect errors and provide their corresponding explanations. This transformation can be described using a *circuit–like representation* of feature models, where each relationship corresponds to a component. Every component or relationship has one binary input per feature and one binary output (see Figure 7). Each input represents the presence (1) or absence (0) of a feature whereas each output represents whether a relationship is satisfied (1) or not (0). A product, represented by its selected features, is an *instance* of the feature model if all relationships are satisfied, i.e. if all outputs are equal to 1.

Figure 7 shows the circuit–like representation of the HIS feature model in Figure 2. For example, the component representing the relationship `R5` has two inputs, `supervision` and `fire`, representing the corresponding features. If both features are present or absent at the same time in a product, the

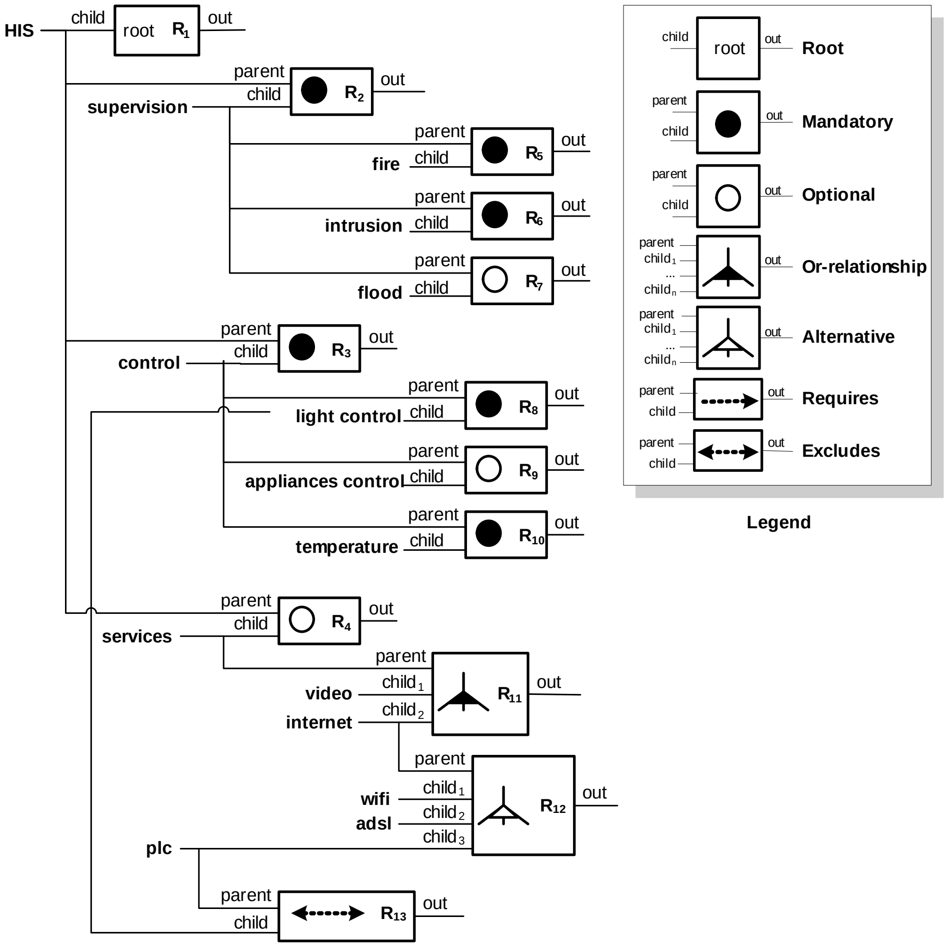


Figure 7. Circuit–like representation of the HIS feature model

`R5` component outputs 1; otherwise it outputs 0, which corresponds to the semantics of mandatory relationships (see Section 2.1).

4.1 Transforming a Feature Model into a Diagnosis Model

As described in section 2.2, *COMPS*, *SD* and *OBS* sets must be defined to represent a feature model as a diagnosis model. In our circuit–like representation of feature models, the relationships are considered as the components of the circuit to be diagnosed. In other words, $COMPS = \{\ R_1, \ldots, R_n\ \}$ where R_i represents the `Ri` relationship in the feature model.

To define the *SD* set, some notation must be previously adopted. All R_i components, except those representing a root relationship, have one *parent* input and one or more *child* inputs (see legend in Figure 7). The expression $parent(R_i)$ denotes the parent input of the R_i component and $child(R_i)$ denotes its child input. Whenever a component has a variable number of children (*or* and *alternative* relationships), the expression $child_j(R_i)$ denotes the j^{th} child input of component R_i, with $1 \leq j \leq m$. For all type of components, $out(R_i)$ denotes their output.

Once the notation is defined, the behavioral model of the diagnosis system can be specified as shown in Figure 8. For the sake of simplicity, all definitions have the form $\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ behaviour_1(R_i)\ )\ )$, where $behaviour_1(R_i)$ is a predicate relating the inputs of the R_i component that must

Type of R_i	**Behavior**
Root	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ child(R_i) = 1\ )\ )$
Mandatory	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ child(R_i) = 1 \Leftrightarrow parent(R_i) = 1\ )\ )$
Optional	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ child(R_i) = 1 \Rightarrow parent(R_i) = 1\ )\ )$
Alternative	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow ($ $(\ parent(R_i) = 1\ \wedge\ \sum_{j=1}^{m} child_j(R_i) = 1\ )\ \vee$ $(\ parent(R_i) = 0\ \wedge\ \sum_{j=1}^{m} child_j(R_i) = 0\ )\ )\ )$
Or	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow ($ $(\ parent(R_i) = 1\ \wedge\ \sum_{j=1}^{m} child_j(R_i) \geq 1\ )\ \vee$ $(\ parent(R_i) = 0\ \wedge\ \sum_{j=1}^{m} child_j(R_i) = 0\ )\ )\ )$
Requires	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ parent(R_i) = 1 \Rightarrow child(R_i) = 1\ )\ )$
Excludes	$\neg Ab(R_i) \Rightarrow (\ out(R_i) = 1 \Leftrightarrow (\ parent(R_i) = 1 \Rightarrow child(R_i) = 0\ )\ )$

Figure 8. Mapping a feature model onto a diagnosis behavioral model

hold when its output is 1. There is no need to include component behaviour when their output is 0 in SD because, since all possible input and output values are $\{ \ 0,1 \ \}$, it can be deduced that $\neg Ab(R_i) \Rightarrow (\ out(R_i) = 0 \Leftrightarrow (\ \neg behaviour_1(R_i) \ ) \ )$.

To complete the definition of SD, the structural model that describes how feature signals and component inputs bind must be defined. As an example, the structural model corresponding to the circuit in Figure 7 is shown in Figure 9.

$child(R_1) = \texttt{HIS}$ [structural model]

$parent(R_2) = \texttt{HIS}$

$child(R_2) = \texttt{supervision}$

$parent(R_5) = \texttt{supervision}$

$child(R_6) = \texttt{fire}$

$\ldots$

$\neg Ab(R_1) \Rightarrow (\ out(R_1) = 1 \Leftrightarrow (\ child(R_1) = 1 \ ) \ )$ [behavioral model]

$\neg Ab(R_2) \Rightarrow (\ out(R_2) = 1 \Leftrightarrow (\ parent(R_2) = 1 \Leftrightarrow child(R_2) = 1 \ ) \ )$

$\neg Ab(R_3) \Rightarrow (\ out(R_3) = 1 \Leftrightarrow (\ parent(R_3) = 1 \Leftrightarrow child(R_3) = 1 \ ) \ )$

$\ldots$

Figure 9. Diagnosis system description corresponding to Figure 7

4.2 Diagnosing a Feature Model

The third element in a diagnosis model is the set of observations. Diagnosing a system relies on consistency checking, i.e. detecting contradictions between the system description and a given set of observations assuming that all components are behaving normally. In the case of diagnosing feature models, we assume that all relationships are satisfied, i.e. $\forall_{i=1}^{n} \ out(R_i) = 1$, and forcing one or more features to be present or absent, as described in the following sections.

4.2.1 Diagnosing Dead Features

A dead feature is a feature that does not appear in any product. In other words, if $\forall_{i=1}^{n} \ out(R_i) = 1$, that feature cannot be present in any input. By translating this concept into a diagnosis model, we can affirm that if the following set of predicates is not consistent, then f_{dead} is a dead feature:

$$SD \ \cup \ \{ \ \forall_{i=1}^{n} \, out(R_i) = 1, \ f_{dead} = 1 \ \} \ \cup \ \{ \ \neg Ab(R_i) \mid R_i \in COMPS \ \}$$

Applying the theory of diagnosis we may determine all possible diagnoses $\{\Delta_1, \cdots, \Delta_k\}$ that make f_{dead} be a dead feature. Each Δ_i is a subset of components, i.e. relationships in the feature model, that makes the following set of predicates consistent:

$$\begin{aligned} SD \ &\cup \{ \ \forall_{i=1}^{n} \, out(R_i) = 1, \ f_{dead} = 1 \ \} \\ &\cup \{ \ Ab(R_i) \mid R_i \in \Delta_i \ \} \\ &\cup \{ \ \neg Ab(R_i) \mid R_i \in COMPS - \Delta_i \ \} \end{aligned}$$

In the `HIS` sample feature model, the `plc` feature is a dead feature because the observation $OBS = \{\forall_{i=1}^{n} \, out(R_i) = 1, \ \texttt{plc} = 1\}$ is not consistent with the system description assuming all components are behaving normally. The reason is that the `plc` feature is incompatible with the `light control` feature, which is a mandatory one. This situation is reflected in the set of diagnoses for that observation, $\Delta_1 = \{R_3\}$, $\Delta_2 = \{R_8\}$ and $\Delta_3 = \{R_{13}\}$, which indicates that relationships `R3`, `R8` and `R13` are responsible of making `plc` a dead feature. If `R3` or `R8` were turned into optional relationships or `R13` were turned into a `requires` relationship or removed, `plc` would become a *live* feature.

4.2.2 Diagnosing Full–Mandatory Features

A full–mandatory feature is a feature that must be present in a product whenever its parent feature is, despite of being a child feature in a non–mandatory relationship, i.e. optional, or–relationship or alternative. Following a similar rationale than for diagnosing dead features, it means that if R_i is a non–mandatory relationship, there cannot be any product in which $parent(R_i) = 1$ and $child_j(R_i) = 0$, being $child_j(R_i)$ the child input of R_i bound to the feature to be checked as full–mandatory. In diagnosis terms, if the following set of predicates is not consistent:

$$\begin{aligned} SD \ &\cup \{ \ \forall_{i=1}^{n} \, out(R_i) = 1, \ parent(R_i) = 1, child_j(R_i) = 0 \ \} \\ &\cup \{ \ \neg Ab(R_i) \mid R_i \in COMPS \ \} \end{aligned}$$

then the feature bound to $child_j(R_i)$ is a full–mandatory feature. The explanations for this kind of error follow the same reasoning as for dead features, i.e. determining the Δ_i diagnoses.

4.2.3 Diagnosing Void Feature Models

A feature model is void if there not exist any product satisfying all its relationships, i.e. if it does not describe any product at all. This situation happens when the root feature is itself a dead feature, so it can be diagnosed following the dead features diagnosis rationale, i.e. if $OBS = \{\forall_{i=1}^{n} out(R_i) = 1,\ child(R_{root}) = 1\}$ makes the system not consistent, the corresponding model is a void feature model.

For the rest of the article, void feature models will be considered as a particular case of dead features and no special treatment will be described.

5 Implementation Level: Modeling Diagnosis Problem as a CSP

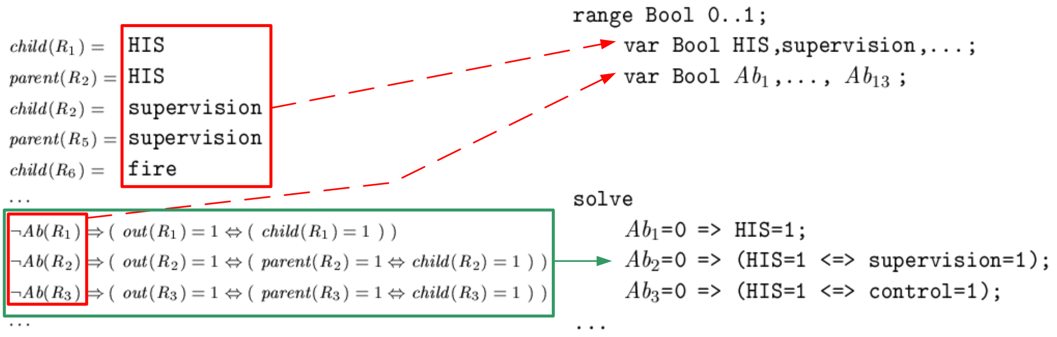
One of the main advantages of defining error detection and explanation in terms of theory of diagnosis is having the problem described in an implementation–independent way. In this work, we propose an implementation based on constraint programming, however any other implementation could be proposed relying on the previous diagnosis level.

Our proposal is inspired by two main sources: on the one hand, Benavides et al. (2005) proposed a direct mapping from a feature model onto a CSP to extract information about them; on the other hand, Fattah and Dechter (1995) proposed a general transformation from diagnosis problems into CSPs.

5.1 Transforming a Diagnosis Model into a CSP

The first step to describe a CSP is determining the set of variables (V) and their domains (D). In our case, we distinguish two kinds of variables defined over domain $\{0, 1\}$: *feature variables*, $V_F = \{F_1, \cdots, F_m\}$, corresponding to the features variables in the structural model of the diagnosis system; and *abnormality variables*, $V_{Ab} = \{Ab_1, \cdots, Ab_n\}$, corresponding to the abnormality indicators in the behavioral model of the diagnosis system.

Notice that there are no variables corresponding to the outputs of the components representing the relationships in the feature models. Since $out(R_i) = 1$ is a condition present in all the observations required for diagnosing a feature model, it can be assumed that it always holds and therefore simplify the behavioral model definitions from $\neg Ab(R_i) \Rightarrow (out(R_i) = 1 \Leftrightarrow behaviour(R_i)\ )$ into $\neg Ab(R_i) \Rightarrow behaviour(R_i)$ and their corresponding constraints in a similar manner (see Figure 10).



$child(R_1) =$ HIS
$parent(R_2) =$ HIS
$child(R_2) =$ supervision
$parent(R_5) =$ supervision
$child(R_6) =$ fire
...
$\neg Ab(R_1) \Rightarrow (out(R_1) = 1 \Leftrightarrow (child(R_1) = 1))$
$\neg Ab(R_2) \Rightarrow (out(R_2) = 1 \Leftrightarrow (parent(R_2) = 1 \Leftrightarrow child(R_2) = 1))$
$\neg Ab(R_3) \Rightarrow (out(R_3) = 1 \Leftrightarrow (parent(R_3) = 1 \Leftrightarrow child(R_3) = 1))$
...

```
range Bool 0..1;
    var Bool HIS,supervision,...;
    var Bool Ab1,..., Ab13 ;

solve
    Ab1=0 => HIS=1;
    Ab2=0 => (HIS=1 <=> supervision=1);
    Ab3=0 => (HIS=1 <=> control=1);
...
```

Figure 10. Transforming diagnosis model in Figure 9 into a CSP

The second step is defining the constraints of the CSP. For that purpose, a straightforward transformation from the structural and behavioral model of the diagnosis system into a set of constraints is performed, as depicted in Figure 10 using *Optimization Programming Language* (OPL), a widely used language to represent constraint programming problems (Hentenryck, 1999).

Notice that predicates of the form $\neg Ab(R_i)$ are translated into a condition on the corresponding abnormality variable of the form $Ab_i = 0$ and that *parent* and *child* expressions are substituted by the corresponding feature variable in the structural model of the diagnosis system.

As previously mentioned in Section 2.3, a solution to a CSP is an assignment of domain values to the variables that makes all the constraints hold. Taking into account that a valid product must satisfy all the relationships in a feature model, the derived CSP can be used to determine the set of valid products defined by a feature model if the values of abnormality variables are all set to zero in the constraints set, i.e. $\{Ab_j = 0 \mid Ab_j \in V_{Ab}\}$. All the solutions to the resulting CSP would be assignments to the feature variables F_i, i.e. valid product configurations in which the assignment $F_i \mapsto 1$ means F_i is present in a product configuration whereas $F_i \mapsto 0$ means that F_i is absent.

5.2 Diagnosing a Feature Model

The key element for diagnosing a system is the set of observations. As previously described in Section 4.2, in order to diagnose a feature model, all relationships are assumed to be satisfied, $\forall_{i=1}^{n} out(R_i) = 1$, and one or more features are forced to be present or absent depending on the kind of error to be diagnosed.

When the diagnosis system is transformed into a CSP, observations become conditions of the form $F_i = 0$ or $F_i = 1$ that are added to the set of constraints, and consistency checking becomes satisfiability checking. In the next sections the additional constraints derived for each kind of error based on the observations proposed in Section 4.2 are described.

5.2.1 Detecting Dead Features

As described in Section 4.2.1, in order to diagnose if a feature f_{dead} is a dead feature in a feature model, the following set of predicates must be checked for consistency:

$$SD \cup \{\ \forall_{i=1}^{n} out(R_i) = 1,\ f_{dead} = 1\ \} \cup \{\ \neg Ab(R_i) \mid R_i \in COMPS\ \}$$

Transforming this into a CSP implies that the following conditions must be added to the set of constraints of the CSP derived from the diagnosis model: $\{Ab_j = 0 \mid Ab_j \in V_{Ab}\} \cup \{F_{dead} = 1\}$. The consistency checking is then replaced by a satisfiability checking, i.e. if the augmented CSP is not satisfiable, f_{dead} is a dead feature.

5.2.2 Detecting Full–mandatory Features

In the case of full–mandatory features, the transformation of the diagnosis system into a CSP implies that the following set of conditions must be added to the set of constraints of the derived CSP: $\{Ab_j = 0 \mid Ab_j \in V_{Ab}\} \cup \{F_{fm} = 0\} \cup \{F_p = 0\}$, where F_{fm} is the full–mandatory feature and F_p it is its parent feature. If the augmented CSP is not satisfiable, F_{fm} is a full–mandatory feature.

5.3 Explaining errors

Once the errors have been detected, their causes must be determined, i.e. which are the relationships generating the errors. In the diagnosis level, a diagnosis is a minimal set of relationships that behave abnormally and that explains the errors in the feature model. Transforming this into a CSP, a diagnosis Δ is a minimal set of abnormality variables $\Delta \subseteq V_{Ab}$ such that $\{Ab_k \mapsto 1 \mid Ab_k \in \Delta\}$ is in the set of solutions of the CSP. In other words, not all the solutions are interesting but only those minimizing the number of failing relationships, i.e. the number of abnormality variables taking value 1. As described in Section 2.3, the CSP for determining Δ sets is a CSOP in which the objective function, $\sum_{k=1}^{m} Ab_k$, must be minimized.

The set of constraints of a CSP for determining the cause of an error is the same than for detecting the error except that abnormality variables are unbound.

For the sample feature model in Figure 2, after detecting that `plc` is a dead feature, the CSOP for providing explanations would be the following:

$$(V_F \cup V_{Ab}, D, C \cup \{\texttt{plc} = 1\}, min \sum_{k=1}^{m} Ab_k)$$

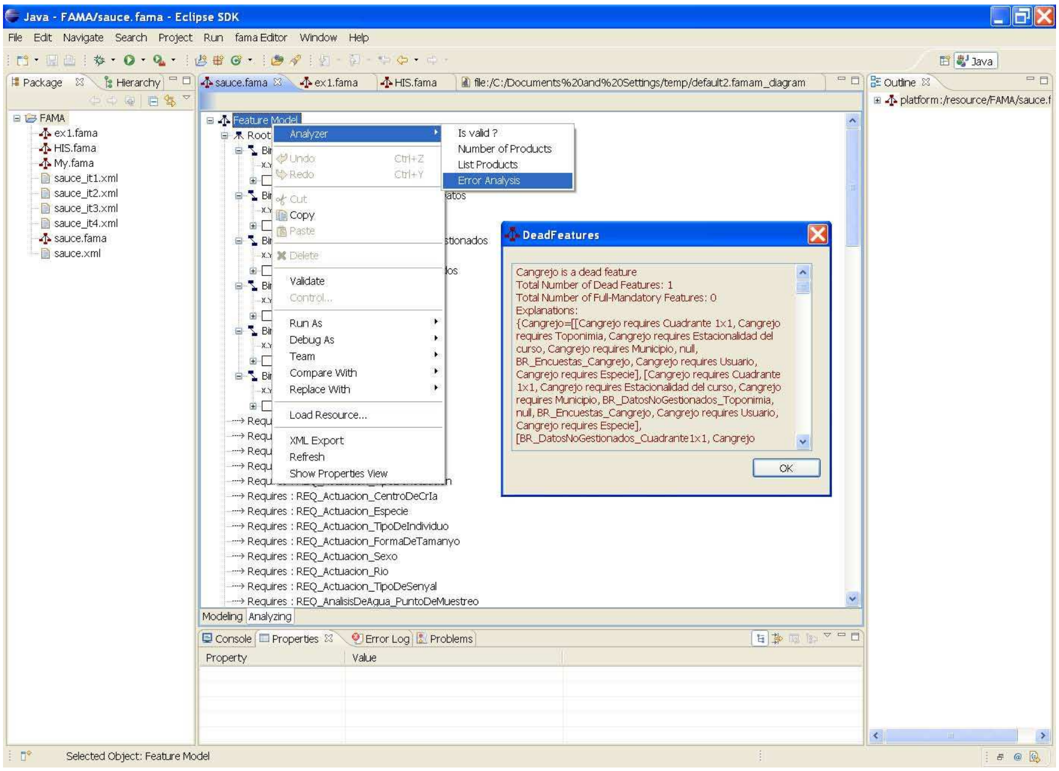
From all its solutions, and after discarding feature variables and abnormality variables taking value 0 —they are not relevant for this purpose—, three Δ set are found: $\{Ab_3 \mapsto 1\}$, $\{Ab_8 \mapsto 1\}$ and $\{Ab_{13} \mapsto 1\}$. Since each abnormality variable is associated with a relationship in the feature model, the three explanations of why `plc` is a dead feature are $\Delta_1 = \{\texttt{R3}\}$, $\Delta_2 = \{\texttt{R8}\}$ and $\Delta_3 = \{\texttt{R13}\}$.

6 Applying our Proposal to a Real Case

We have applied the implementation level described in this paper during a software product line development project. The project intends to build a set of *Enterprise Resource Planning* (ERP) products in the context of SAUCE, an environmental resources management SPL. SAUCE comprises a set of products to store and exploit the existing information about flora and fauna in different rivers. The aim of the SPL is to produce customized software that helps to the management and conservation of these fluvial ecosystems.

As a first result of domain engineering activities, a large–scale feature model was obtained. We followed an approach inspired by the FDD methodology to develop and refine the ERP feature model in two–week iterations. In each iteration the feature model was reviewed according to changes suggested by domain and application engineers using our FAMA tool described by Benavides et al. (2007). FAMA is an Eclipse plugin for feature model edition and analysis. It has a multisolver analysis engine that performs operations such as products counting, products filtering and commonality analysis by means of different CSP, BDD or SAT solvers. FAMA has been extended to support the error analysis implementation described in this article. This way, FAMA assists the production of error–free feature models by detecting and explaining the emerging errors. Some captures of the process are shown in Figure 11. The empirical results obtained from FAMA in each iteration are presented in Figure 12. Each row corresponds to an iteration. Five iterations were performed over the feature model and their data collected. The columns are labeled as follows:

- #F: number of features of the feature model.
- #R: number of relationships without considering cross–tree relationships, i.e. *requires* and *excludes* relationships.



Java - FAMA/sauce.fama - Eclipse SDK
File Edit Navigate Search Project Run fama Editor Window Help
Package
Hierarchy
FAMA
ex1.fama
HIS.fama
My.fama
sauce_it1.xml
sauce_it2.xml
sauce_it3.xml
sauce_it4.xml
sauce.fama
sauce.xml
sauce.fama
ex1.fama
HIS.fama
file:/C:/Documents%20and%20Settings/temp/default2.famam_diagram
Feature Model
Root
Analyzer
Undo Ctrl+Z
Redo Ctrl+Y
Cut
Copy
Paste
Delete
Validate
Control...
Run As
Debug As
Team
Compare With
Replace With
Load Resource...
XML Export
Refresh
Show Properties View
Is valid ?
Number of Products
List Products
Error Analysis
Requires : REQ_Actuacion_CentroDeCrIa
Requires : REQ_Actuacion_Especie
Requires : REQ_Actuacion_TipoDeIndividuo
Requires : REQ_Actuacion_FormaDeTamanyo
Requires : REQ_Actuacion_Sexo
Requires : REQ_Actuacion_Rio
Requires : REQ_Actuacion_TipoDeSenyal
Requires : REQ_AnalisisDeAgua_PuntoDeMuestreo
Modeling
Analyzing
Console
Properties
Error Log
Problems
Property
Value
Outline
platform:/resource/FAMA/sauce.f
Java
Selected Object: Feature Model
DeadFeatures
OK

```
Cangrejo is a dead feature
Total Number of Dead Features: 1
Total Number of Full-Mandatory Features: 0
Explanations:
{Cangrejo=[[Cangrejo requires Cuadrante 1x1, Cangrejo
requires Toponimia, Cangrejo requires Estacionalidad del
curso, Cangrejo requires Municipio, null,
BR_Encuestas_Cangrejo, Cangrejo requires Usuario,
Cangrejo requires Especie], [Cangrejo requires Cuadrante
1x1, Cangrejo requires Estacionalidad del curso, Cangrejo
requires Municipio, BR_DatosNoGestionados_Toponimia,
null, BR_Encuestas_Cangrejo, Cangrejo requires Usuario,
Cangrejo requires Especie],
[BR_DatosNoGestionados_Cuadrante1x1, Cangrejo
```

Figure 11. FAMA tool has supported the error detection and explanation in SAUCE development

- #CTR: number of cross–tree relationships.
- #FC: number of added plus removed features respecting to the previous iteration.
- #RC: number of changes affecting the existing relationships plus number of added and removed relationships respecting to the previous iteration.
- #CTRC: number of added plus removed cross–tree relationships respecting to the previous iteration.
- #P: approximate number of products represented by the feature model.
- #DF: number of dead features detected by our tool and corrected by the user.
- VFM: if the resulting feature model included an error to make it to be void.
- #FMF: total number of full–mandatory features detected by our tool and corrected by the user.

In each iteration, as a consequence of the changes in the SPL requirements and therefore in the feature model, new errors arose even when the previous feature model was error–free.

As a result of our experience, a high number of cross–tree relationships often hinder the engineers to keep a record of the arising errors. Our tool has supported the evolution of the ERP feature model easing and guaranteeing the production of an error–free feature model. Supporting a quicker evolution of

It	#F	#R	#CTR	#FC	#RC	#CTRC	#P	#DF	VFM	#FMF
1	61	56	54	-	-	-	$3,59 \cdot 10^{10}$	1	No	2
2	76	70	86	15	14	32	$2,96 \cdot 10^{13}$	1	No	8
3	79	73	88	3	3	2	$1,17 \cdot 10^{14}$	0	No	0
4	84	78	102	5	9	14	$5,18 \cdot 10^{14}$	2	No	4
5	86	80	104	2	2	3	$1,46 \cdot 10^{16}$	0	No	6

Figure 12. Evolution of the feature model of an ERP SPL

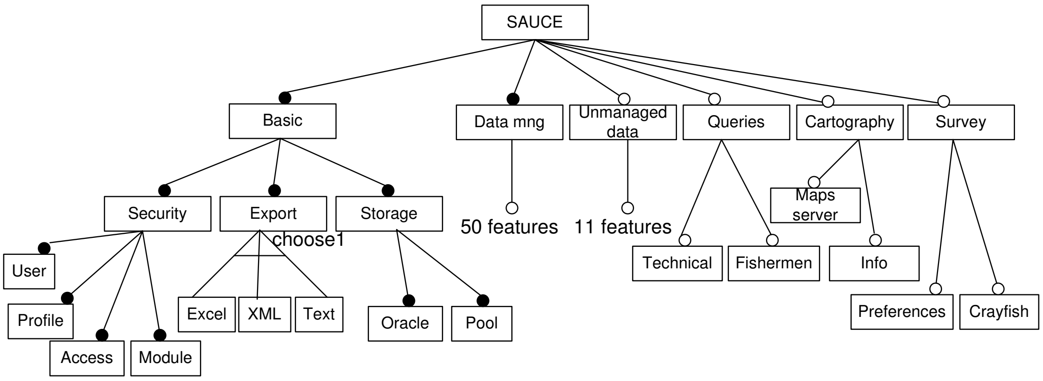


Figure 13. ERP feature model

feature models reduces the time invested on this task, allowing the engineers to concentrate in others. The feature model resulting from this process is depicted in Figure 13. Notice that for the sake of simplicity we have omitted part of the features on the feature model and all the cross–tree relationships.

7 Related Work

Although the automated error analysis in feature models was already identified as a fundamental task in the original FODA report by Kang et al. (1990), few authors have dealt with it. As a matter of fact, there has not been a seminal approach to automatically analyze errors in feature models as far as we know.

Our interest in automating error detection and explanation arose from the work of von der Massen and Lichter (2004), where the authors proposed a categorization of what they call *deficiencies* (referred to as *errors* in this article) in three levels of severity: redundancy, anomaly and inconsistency. Redundancies appear when relationships among features are modeled in multiple ways so they can be removed and the set of products represented by a feature model remains the same. In some cases, redundancies can be intentionally introduced to emphasize a relationship. We have not dealt with them in this article because they do not fit in our concept of error. Anomalies appear when some products are lost due to a mismodelling but the feature model still defines some products. Anomalies generate dead and full–mandatory features. Finally, inconsistencies appear when the feature model contains contradictory relationships removing a set of products (dead features) or making it impossible to derive products (void feature models). Unfortunately, von der Massen and Lichter's proposal lacks rigorous definitions and no automated analysis is suggested.

Regarding the works dealing with automated error analysis, we distinguish between those that only deal with error detection and those also coping with errors explanation. In the first group we mention the work of Mannion (2002); Zhang et al. (2004); Czarnecki and Kim (2005).

Mannion uses first–order logic to determine if a feature model is void or not, but no other kind of error is detected. Zhang et al. suggest the use of an automated tool support based on the SVM System (McMillan, 1992) to detect void feature models and dead features. Finally, Czarnecki and Kim propose the detection of void feature models and dead features as a marginal result of applying binary decision diagrams to represent feature models.

In the second group where errors explanation is dealt with, Batory (2005); Sun et al. (2005); Wang et al. (2005) work on automated error explanation but they are only able to detect whether a feature model is void or not and which are the conflicting relationships.

Batory translates feature models into propositional formulas and uses SAT solvers (solvers for propositional calculus) and *Logic Truth Maintenance Systems* (LTMS) algorithms. Sun et al. translates feature models into *Alloy*, a simple structural modeling language based on first–order logic (Jackson, 2002). Alloy uses a SAT solver to analyze the relationships that generate a void feature model. Finally, Wang et al. propose the translation of feature models into an *OWL DL* ontology. OWL DL is a expressive yet decidable sublanguage of *OWL* (Ontology Web Language). It is possible to use automated tools such as RACER, proposed by Haarslev and Moller (2001) and used in this case to the automatically analyze feature models. A summary of the reviewed proposals is presented in Figure 14.

Although not all the previous proposals allow the analysis of dead and full mandatory features, this is not their main drawback because it is certainly possible to extend them. In our opinion, the main disadvantage of these proposals is that they lack abstraction. It is in the sense that they are useful when feature models are analyzed using the corresponding formalisms and tools but they are not extrapolatable to other ways of analyzing errors in feature mod-

	Mannion (2002)	Czarnecki and Kim (2005)	Batory (2005)	Sun et al. (2005)	Wang et al. (2005)	Zhang et al. (2004)	**Our proposal**
	Operations supported						
Void FMs	+	+	+	+	+	+	+
Dead Features	-	+	-	-	-	+	+
Full Mandatory Features	-	-	-	-	-	-	+
Explanation	-	-	+	+	+	-	+
	Characteristics						
Abstraction	-	-	-	-	-	-	+
Basic FMs	+	+	+	+	+	+	+
Extended FMs	-	+	-	-	-	-	+

Figure 14. Summary of proposals for the automated error analysis of feature models

els. By contrast, in this paper, we have presented a more abstract proposal, because we use theory of diagnosis principles, a well–established research field with strong theoretical foundations, as a more abstract level of specifying the analysis of errors in feature models.

Due to its level of abstraction, our proposal allows extensions in both diagnosis and implementation levels. Other errors can be added in the diagnosis level and implemented in the implementation level that can also be defined using other tools such as *Binary Decision Diagrams* (BDD) or SAT solvers instead of CSP solvers.

New kinds of error can appear when dealing with extended feature models (Benavides et al. (2005); Batory (2005); Batory et al. (2006)) where feature attributes are included in the model. Relationships among attributes can also constrain the model, producing dead features for example. Benavides et al. (2005) proposed a direct mapping from a feature model onto CSP that represents attributes. As we have proposed a general schema that supports new errors just by defining the observation that detects them in the diagnosis level, and an implementation to deal with attributes already exists, we think that we can extend our proposal to support errors analysis in extended feature models. This is an important limitation of the other proposals.

8 Conclusions and Future work

We have discussed our vision on how SPL and agile methods can come together, either by applying agile principles to SPL methodologies or by tailoring an existing agile methodology to support SPL development. Independently from the chosen alternative, supporting automatic error detection and explanation is an important contribution that can be a first step in bringing agile principles and SPL together. As feature modeling is an error–prone task, an activity that checks the feature model is needed. Large–scale feature models may contain hundreds of features and represent thousands of products as it can be seen in our example case. In these cases, an automated support for error analysis is needed as doing it by hand is not feasible. Our proposal supports an automated feature model error analysis that is therefore, a first step in our roadmap to integrating agile and SPL techniques.

Our proposal relies on theory of diagnosis to represent the problem of error detection and explanation in general terms. The advantage of using this abstract representation is twofold: many different implementations can be used in the implementation layer and the diagnosis level can be extended with new kinds of error just by defining the observations that detected them.

By relying on the extensibility of our proposal we have detected some future extensions to our proposal. In programming languages, errors correction is a mature topic. Explanations give the user sufficient information to correct errors. We will study how to use explanations to assist the user with the correction of errors.

As several implementations can be used to refine the diagnosis level, it is important to compare how each implementation performs to choose the best of them. Our future work will compare the performance of several SAT, CSP and BDD solvers.

We have only focused on errors analysis in basic feature models, but they can be extended with attributes where new kinds of error can appear. Our future work will extend our proposal to deal with extended feature models.

Regarding the integration of SPL and agile methods integration we would like to thoroughly study both alternatives. Specifically, we plan to tailor all the stages in FDD to fully support SPL based on feature models.

Although we have presented the most used notation of feature models, it is important to notice that there are other notations with different semantics as described by Schobbens et al. (2007). An unified language for feature modeling is needed and if this language is adopted our idea will remain valid but we may have to change the mapping from this new language to theory of diagnosis.

Acknowledgments

We would like to thank Richard Page, Patrick Heymans and Pierre–Yves Schobbens for their useful comments and Manuel Nieto for his effort on applying our ideas in an industrial case study.

References

[1] Batory, D., 2005. Feature models, grammars, and propositional formulas. In: Software Product Lines Conference, LNCS 3714. pp. 7–20.

[2] Batory, D., Benavides, D., Ruiz-Cortés, A., 2006. Automated analysis of feature models: Challenges ahead. Communications of the ACM 49 (12), 45–47.

[3] Batory, D., Sarvela, J., Rauschmayer, A., 2004. Scaling step-wise refinement. IEEE Trans. Software Eng. 30 (6), 355–371.

[4] Benavides, D., Ruiz-Cortés, A., Trinidad, P., 2005. Automated reasoning on feature models. LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005 3520, 491–503.

[5] Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2007. FAMA: Tooling a framework for the automated analysis of feature models. In: Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS). pp. 129–134.

[6] Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., Pietroszek, K., 2005. Model-driven software product lines. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM Press, New York, NY, USA, pp. 126–127.

[7] Czarnecki, K., Kim, P., 2005. Cardinality-based feature modeling and constraints: A progress report. In: Proceedings of the International Workshop on Software Factories At OOPSLA 2005.

[8] de Kleer, J., Mackworth, A. K., Reiter, R., 1990. Characterizing diagnoses. In: AAAI. pp. 324–330.

[9] Fattah, Y. E., Dechter, R., 1995. Diagnosing tree-decomposable circuits. In: IJCAI. pp. 1742–1749.

[10] Fowler, M., Highsmith, J., August 2001. The agile manifesto. In Software Development, Issue on Agile Methodologies.

[11] Haarslev, V., Moller, R., 2001. Description of the RACER system and its applications. In: Description Logics.
URL `http://www.racer-systems.com`

[12] Hentenryck, P. V., 1999. The OPL optimization programming language. MIT Press, Cambridge, MA, USA.

[13] Jackson, D., 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11 (2), 256–290.

[14] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., Nov. 1990. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

[15] Mannion, M., 2002. Using First-Order Logic for Product Line Model Validation. In: Proceedings of the Second Software Product Line Conference (SPLC2). LNCS 2379. Springer, San Diego, CA, pp. 176–187.

[16] McMillan, K. L., 1992. The svm system.
URL `http://www.cs.cmu.edu/∼modelcheck/smv.html`

[17] Pohl, K., Böckle, G., , van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer–Verlag.

[18] Raatikainen, M., Soininen, T., Männistö, T., Mattila, A., 2005. Characterizing configurable software product families and their derivation. Software Process: Improvement and Practice 10 (1), 41–60.

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

[20] Schobbens, P., P. Heymans, J. T., Bontemps, Y., Feb 2007. Generic semantics of feature diagrams. Computer Networks 51 (2), 456–479.

[21] Sochos, P., Philippow, I., Riebisch, M., 2004. Feature-oriented development of software product lines: Mapping feature models to the architecture. Object-Oriented and Internet-Based Technologies, 138–152.

[22] Sun, J., Zhang, H., Li, Y., Wang, H., 2005. Formal semantics and verification for feature modeling. In: Proceedings of the ICECSS05.

[23] Tsang, E., 1995. Foundations of Constraint Satisfaction. Academic Press.

[24] von der Massen, T., Lichter, H., 2004. Deficiencies in feature models. In: Mannisto, T., Bosch, J. (Eds.), Workshop on Software Variability Management for Product Derivation - Towards Tool Support.

[25] Wang, H., Li, Y., Sun, J., Zhang, H., Pan, J., November 2005. A semantic web approach to feature modeling and verification. In: Workshop on Semantic Web Enabled Software Engineering (SWESE'05).

[26] Williams, L., 2004. A survey of agile development methodologies.
URL `http://agile.csc.ncsu.edu/SEMaterials/AgileMethods.pdf`

[27] Zhang, W., Zhao, H., Mei, H., 2004. A propositional logic-based method for verification of feature models. In: Davies, J. (Ed.), ICFEM 2004. Vol. 3308. Springer–Verlag, pp. 115–130.