Prototyping Software Product Lines analysis with Pharo

Abdelghani Alidra
Science Faculty,
20 Aout 55 University, Skikda
Algeria
alidrandco@yahoo.fr

Mohamed Tahar Kimour
LASE Laboratory,
Badji Mokhtar University,
Algeria
mtkimour@hotmail.fr

Abstract

Software Product Lines (SPLs) are an emerging software engineering paradigm that aims to optimize software development costs and time to market trough systematic development of reusable of core assets. At the heart of SPL engineering is variability modelling. Feature models (FM) are a common way to model variability and reason about it. Examples of reasoning are for instance checking that at least one product is represented by a given FM (satisfiability) or finding the product that best fits a given set of requirements. In practice however, such operations are often complex and time consuming. In order to address these challenges, we introduce in the present article the notion of transitive dependency between features and show how it can be used as the basis for efficient analysis and automatic reasoning on feature models. We exploit this new concept to implement a first platform for prototyping and reasoning on large SPLs in Pharo. Finally we illustrate the efficiency of our proposal on the problem of features selection optimisation.

Keywords Software Product Lines, Feature models, automatic reasoning, analysis environment

1. Introduction

Software Product Lines (SPLs) are an emerging software engineering paradigm that aims to optimize software's development costs and time to market trough systematic development and exploitation of reusable core assets.

Product line engineering is a concept comprising methods, tools, and techniques for the development of product lines [1][6][7]. A software product line is the process aiming at designing and managing a set of related software products that target a specific domain and share some parts of their code. By selecting varying sets of assets, different products (aka variants), fulfilling different requirements of a specific application, can be generated. Software product lines engineering has attracted considerable research effort and shown to bring tremendous improvement in software development costs and time-to-market [1]. At the heart of SPL engineering is variability modelling. Feature models (FM) are a common way to model variability and reason about it.

A central idea in SPL Engineering is the explicit modelling of commonalities and variations between the family software products [4]. Feature Models [3][4][5] are frequently used to this end. A feature model represents the hierarchical arrangement of system functionalities and the relationships among them which constrains the valid composition of functionalities defining the products of the software family.

An important advantage of using feature models is the capacity of automatic reasoning on the valid combination of features (and thus core assets). For instance, a stakeholder may want to check the satisfiability of the feature model, that is, check whether at least one product is derivable from the product line. Another example is identifying the derivable products, calculating the number of derivable products, checking whether a given product belongs to the software family, identifying the product optimizing the stakeholder requirements…

In practice however, such operations are often difficult to implement and time consuming [9][10][11][12][13][14][18]. This is especially true for a number of real world systems with many features (hundreds or even thousands). For instance, finding the optimal Linux configuration (the Linux kernel has a 6888 features) may take from tens of minutes to hundreds of hours [18]. The introduction of dynamic software product lines, where the automatic reasoning is delayed to runtime to implement adaptable systems, makes this limitation even more critic [15][16][17][8].

To overcome these limitations, we introduce the notion of *transitive dependencies* between features. Indeed, features in the FM depend on each others. A feature depends on another feature if the inclusion (respectively, the exclusion) of the first one implies the inclusion or exclusion (respectively the exclusion) of the second one. For instance, if A is a sub feature of B then, the inclusion of feature A in a given product implies the inclusion of B. Moreover, the exclusion of feature B implies the exclusion of A. Furthermore, dependencies between features are transitive. For instance, if feature A implies feature B, then A implies all the features that are implied by B and excludes all those that B excludes. This means that the inclusion of feature A can impact a large number of other features resolving much of the variability of the FM and consequently reducing the complexity of the analysis problem.

In order to exploit the notion of transitive selection, we also introduce the *transitive selection operator* which intuition is the propagation of feature's selection to all it dependencies. We then employ this operator to implement a number of reasoning algorithms representing a first step towards a platform for efficient analysis of large SPLs in Pharo.

In order to access the benefits of our proposal, we illustrate the use of the transitive selection operator on the problem of optimal features selection. The algorithm based on the new introduced operator is shown to be up to 90% faster than one of

the most effective algorithms for optimal features selection in the literature [11].

The rest of this article is organized as follows: section 2 remind the main concepts relative to software product lines and feature modelling and reviews the most common operations for automated analysis of feature models. Section 3 introduces the dependency relationship between features and some algorithms for automated analysis of feature models. We present the genetic algorithm for products derivation and compare its performances to the one from the literature in section 4. Some related works are discussed in section 5 and section 6 concludes the paper and highlights future works.

2. Background

2.1 Software product lines and feature modeling

Software product line engineering (SPLE) is a software engineering paradigm comprising methods, techniques and tools for developing software intensive systems that share a set of common functionalities and that are intended for a specific market segment. [30]

Recent works have shown that SPLE can significantly improve software development in terms of software quality, time-to-market and production costs. This is made possible because of the systematic approach to core assets reuse. These advantages make SPLE particularly attractive for the development of software based systems such as smart-homes, cell-phones or automotive systems.

The central idea behind SPLE is the reusability of core assets. The general process is separated in two main phases: "Domain Engineering" which aims at developing the core assets for the software family (that is a common architecture, requirements and reusable components) and "Application Engineering", consisting in the derivation of the final products that meet the specific requirements of customers by exploiting the core assets developed during the first phase.

At the heart of SPLE is the management of commonalities and variations between products. Feature models are one of the most popular tools to document the commonalities and variations throw the concept of features [31][32][33]. A software feature is "a distinguishing characteristic of a software item" [30]. A feature model represents the hierarchical arrangement of the system functionalities and the relationships among them. Relationships between features determine valid compositions. Additional information may be included concerning trade-offs, rationale, and justifications for features selection. Feature modeling has attracted increasing research effort and provides good tool support for automatic reasoning and verification [20]

Figure.1 depicts a possible (much simplified) feature model of smart-phones product family. The feature model of figure.1 is a hierarchical representation of variability relationships between features, variability relationships are for instance: mandatory, optional, single and multiple-choice. Cross-tree constraints can also be modeled such as excludes and requires.

For instance, all smart-phones need to include the two components: hardware and software. Hardware includes a processor, a screen, a RAM memory and sometimes a 3G connector and a GPS. Software would imply an OS whether Windows-Phone or a Linux, a set of applications amount of a specific control application that can be a Win32 (that requires running on Windows) or a Posix application (that requires running on Linux).

For more details on FM notations semantic, reader can refer to [31][33].

In the recent years, much works explored the idea of exploiting the Software Product Line paradigm to handle systems online adaptation. This is known as Dynamic Software Product Lines [34]. More specifically, A DSPL uses feature modeling to represent and reason on the system's variants (configurations) at execution time taking advantage of existing model-based techniques and tools to achieve dynamic adaptation. While promising, this trend puts even more

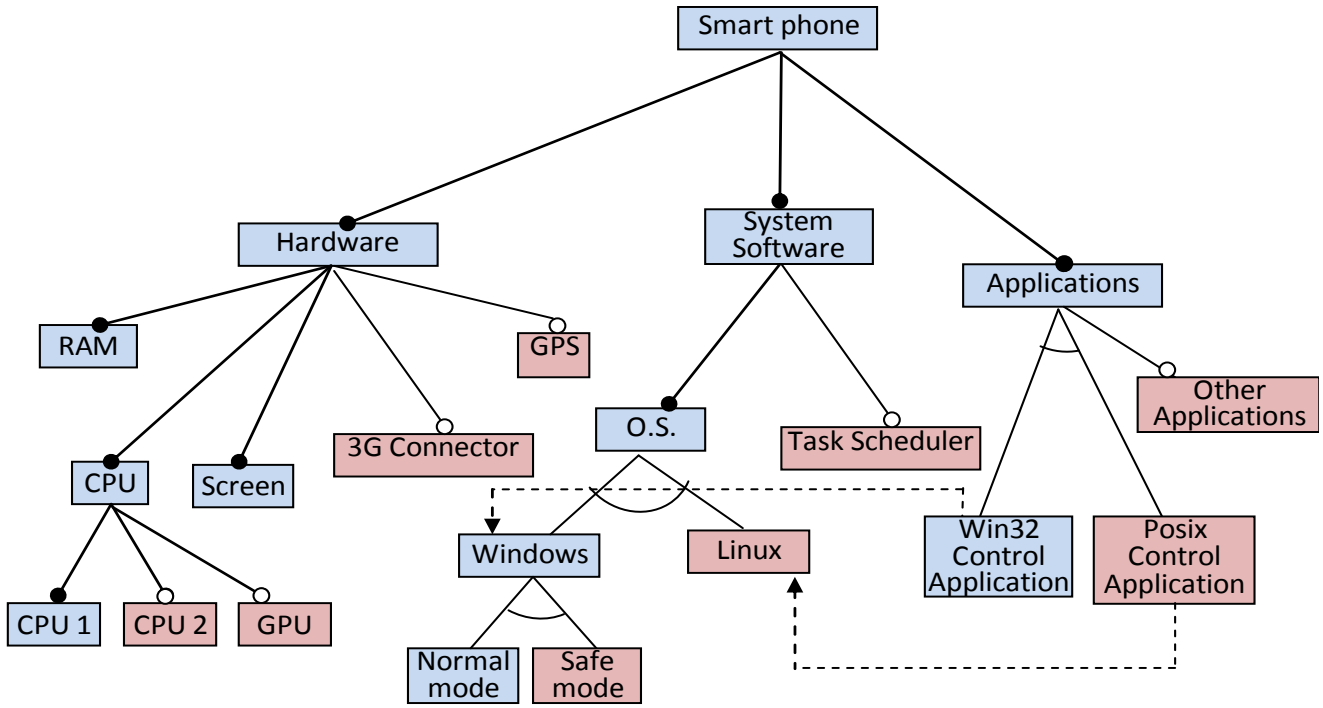


Figure. 1 A sample feature model for the smart-Phones software family

constraints on the automatic operations because of the limited resources and response time.

2.2 Automated reasoning on feature models

As mentioned earlier, the automated reasoning on feature models is a key capability for studying, developing or using software product lines. Automated reasoning allows researchers to study, prototype and develop new SPL techniques. It allows SPL developers define and verify the scope of the product family they work on and finally, it assists stakeholders in deriving the product that best fit their requirements.

Benavides et al surveyed the automated operations on feature models in [19] and [20]. In the following we review the most commonly used ones.

1. **Feature models validation (satisfiability)**: satisfiability is the quality of a feature model from which at least one product can be derived. Because of the inclusion of cross-tree constraints, an FM may be unsatisfiable: it does not accept any derivable products. This may happen if two mandatory features are exclusives.
2. **Calculating the scope of the feature model**: This operation returns the best of all possible products that can be derived from the FM. A subsequent operation is the calculation of the number of derivable products which increases the flexibility and the complexity of the product line.
3. **Calculating commonality**: the commonality of a feature is the percentage of products where this features is present. It reveals information about the importance of the feature in the product line.
4. **Optimizing**: this operation aims at calculating the product(s) that best fit stakeholder's requirements. Stakeholder's satisfiability is described in terms of one or many criterion expressed as fitness functions. A utility (or cost) value is associated to every feature or to combinations of features and the objective is to maximise the overall fitness value. A typical example of such objectives is the cost or energy consumption (to minimize). This operation is NP hard.
5. **Decision Propagation**: Typically, in the process of products derivation, features are selected (or deselected) gradually until a final product is obtained (also known as variability resolution). This process can be either automatic or manual. In both cases, it is important to ensure that features selection respects the FM structural and cross-tree constraints. To do so, every selection (or deselection) must be automatically propagated to the rest of the FM features.
6. **Dead features detection**: Dead features are those that will never appear in any derivable products. Dead features are due to cross-tree constraints. For instance a feature is considered dead if it is part of an exclusion cross-tree constraint together with the root feature.
7. **Providing explanation**: When a feature model is unsatisfiable or when dead features are detected, it is desirable to provide some information about the source of the problem (*i.e.* the cross-tree constraint). A more appreciable (and rare) option is to provide a corrective explanation to fix the problem.
8. **Product validation**: This operation answers the question whether a given product (features combination) is valid against the feature model structural and cross-tree constraints. Here again, an explanation in case the product does not satisfy the feature model and a corrective suggestion are highly appreciable.
9. **Generation of random feature models**: In order to test new proposed operations and techniques, it is often useful to apply them to feature models with different characteristics. This operation is however not trivial since the generated feature model must have some properties essentially the satisfiability and the absence of dead features. This operation can be very complex and time consuming especially if the ratio of cross-tree constraints is high.

In the next section, we introduce the notion of dependency between features as well as the transitive selection operator that relies on it. We will then explain how these concepts can be exploited to performed a large number of the afore mentioned operations

3. Features Dependencies and Transitive Selection.

In order to introduce the notion of features' dependencies, let us first recall the different kinds of constraints in classical feature models. Les us have for notation: $x_i = 0$ means feature i is deselected. $x_i=1$ meaning that feature i is selected.

Thus, we can express the different kinds of constraints as follows:

- **Paternity** : *Feature* $_j$ is the father of *feature* $_i$ meaning that $feature_j$ must be selected whenever $feature_i$ is selected : $x_i=1 => x_j=1$
- **Mandatory** : $Feature_j$ is a mandatory sub feature of $feature_i$. Then, whenever $feature_i$ is selected, $feature_j$ must be selected too : $xi=1 => xj=1$.
- **Alternative** : supposing *features* $_1$.. *feature* $_n$ belong to an alternative group. Then only one feature $feature_i$ may be selected in the group : $xi=1=>xj=0$.
- **sub features** : $Feature_i$ is the parent feature of $features_1..feature_n$ meaning that whenever $feature_i$ is selected, there must be at least one sub feature $feature_j$ selected : $xi=1=>\exists j=1..n/xj=1$
- **Imply** : $feature_i$ implies $feature_j$ meaning that $feature_j$ must be selected whenever $feature_i$ is selected : $xi=1=>xj=1$.
- **excludes** : $feature_i$ excludes $feature_j$ meaning that whenever $feature_i$ is selected $feature_j$ must be deselected and reciprocally : $xi=1=>xj=0$ *and* $xj=1=>xi=0$.

Definition.1: a feature $feature_i$ is said to be dependent of another feature *feature* $_j$, if and only if setting $feature_i$ implies setting $feature_j$, A feature is said to be set if it has a known state whether selected or deselected.

From the above, we can distinguish four kind of dependencies between features:

- **implying** : *feature* $_i$ implies *features* $_j$ if and only if *feature* $_j$ is selected every time *feature* $_i$ is selected : *feature* $_j$ is the parent of *feature* $_i$, *feature* $_j$ is a mandatory sub feature of *feature* $_i$ or *feature* $_i$ requires *feature* $_j$ according to a cross-tree constraint. Note that implying is transitive: if *feature* $_i$ implies *feature* $_k$ and *feature* $_k$ implies *feature* $_j$ => *feature* $_i$ implies *feature* $_j$.
- **exclusion** : *feature* $_i$ excludes *feature* $_j$ if and only if : *feature* $_j$ is deselected every time *feature* $_i$ is selected : *feature* $_i$ and *feature* $_j$ belong to the same xor group or *feature* $_i$ excludes

feature $_j$ according to a cross-tree constraint. Note that exclusion is reflexive: *feature* $_i$ excludes *feature* $_j$ <=> *feature* $_j$ excludes *feature* $_i$. Note also that if *feature* $_i$ requires *feature* $_k$ and *feature* $_k$ excludes *feature* $_j$ => *feature*$_i$ excludes *feature* $_j$.

- **impliedBy** : *feature* $_i$ is impliedBy *feature* $_j$ if and only if *feature* $_j$ implies *feature* $_i$. Note that impliedBy is transitive : if *feature* $_i$ is impliedBy *feature* $_k$ and *feature* $_k$ is impliedBy *feature* $_j$ => *feature* $_i$ is impliedBy *feature* $_j$. this means also that if *feature* $_i$ is deselected then *feature* $_j$ must be deselected too (otherwise, an FM constraint will be violated).
- **subFeature** : this is trivial form the feature model hierarchical structure.

Further, we can define for every *feature*$_i$ the sets ***inclusions***, ***exclusions*** and ***impliedBy*** as follows:

- ***inclusionsOf: feature*$_i$** = {*feature*$_j$*/ feature*$_i$ -> *feature*$_j$ | *feature*$_j$ *isTheParentOf: feature*$_i$| *feature*$_j$ *isMandatorySubFeatureOf: feature*$_i$ | ∃*k/ feature*$_j$ ∈ *inclusionsOf: feature*$_k$ *and feature*$_k$ ∈ *inclusionsOf: feature*$_i$}
- ***exclusionsOf: feature*$_i$** = { *feature*$_j$*/ feature*$_i$->!*feature*$_j$ | *feature*$_i$ *xor feature*$_j$ | ∃*k/feature*$_k$ ∈ *inclusionsOf: feature*$_i$ *and feature*$_j$ ∈ *exclusionsOf: feature*$_k$}
- ***impliedByOf: feature*$_i$** = {*feature*$_j$*/ feature*$_j$->*feature*$_i$, *or feature*$_i$ *isMandatorySubFeatureOf: feature*$_j$ *or feature*$_j$ *isSubFeatureOf: feature*$_i$ *or* ∃*k/feature*$_j$ ∈ *impliedByOf: feature*$_k$ *and feature*$_k$ ∈ *impliedByOf: feature*$_i$}
- ***childrenOf: feature*$_i$** can be naturally defined as {*feature*$_j$*/feature*$_i$ *is parentOf: feature*$_j$}.

3.1 Calculating Dependencies between Features

Figure 2 represents the general overview of the proposed platform for reasoning on large product lines and its implementation in Pharo. The proposed platform is based on Moose, an open source project for agile reengineering[1]. The blue boxes in figure 2 represent existing projects. Green boxes represent tools that we have already developed whereas purple boxes represent future works.

3.1.1 Data Structures

For representing Feature Models, we reused and extended the code of the FAMILIAR project [2]. FAMILIAR is a platform that supports decomposition of complex SPLs by providing visualisations of internal and external constraints between FMs [41]

We particularly extended the package with methods to generate and access features dependencies as follows:

```
FMFeatureModel>>generateDependencies
fmEncoder:= FMEncoder new
            generateDependenciesWith: self.
```

FMEncoder essentially maintains a dictionary associating to every feature its' dependencies. Then, accessing features' dependencies is performed the following operations :

`implyingsOf:aFeature, returns the features implied by aFeature.`

`exclusionsOf:aFeature, returns the features excluded by aFeature.`

`implyedByOf:aFeature, returns the features that imply aFeature.`

Here, a feature dependency (as well as features combinations, i.e. products) is encoded as an instance of the *BitSet* class. At the present moment, *BitSet* is just implemented as a subclass of *Array* whom elements are binary values. Therefore,

aBitSet at:i = 1 means that *featurei* is contained in the set represented by *aBitSet*.

Additionally, *BitSet* implements the following methods:

- *bitUnion:aBitUnion*. Updates the receiver to encode the union of the sets represented by the receiver and the argument. Answers *true* if the receiver has been changed.
- *bitIntersection:aBitUnion*. Updates the receiver to encode the intersection of the sets represented by the receiver and the argument. Answers true if the receiver has been changed.

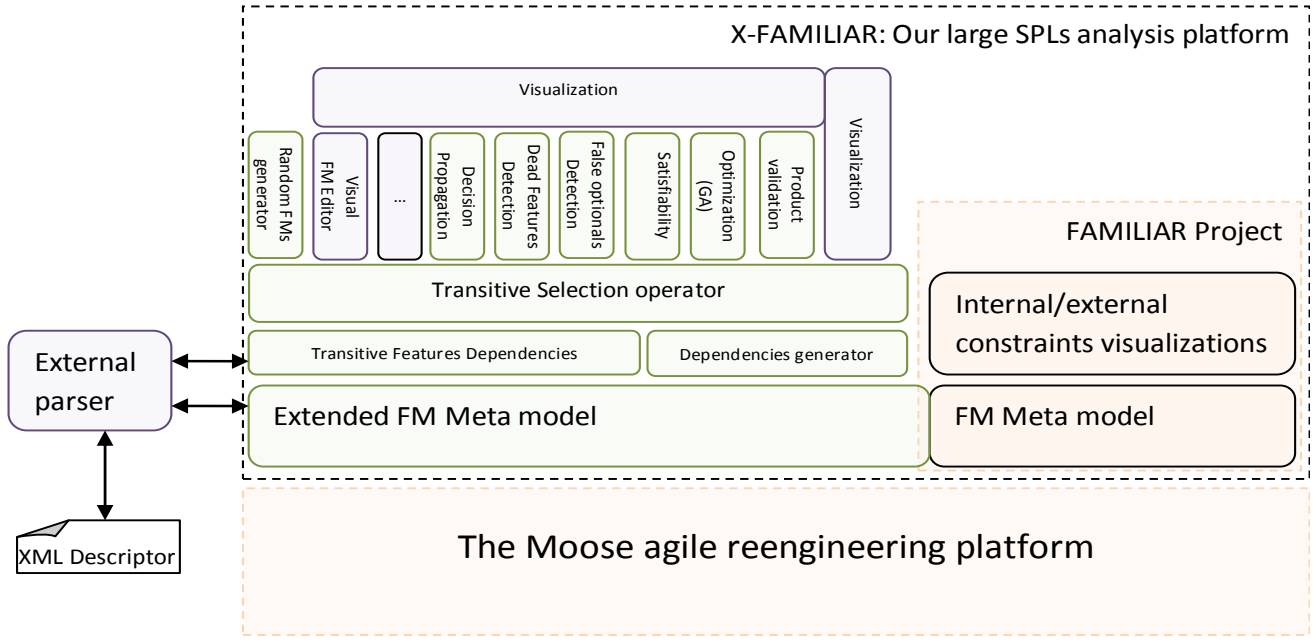


Figure 2. The general architecture of large SPLs analysis platform

- *asCollectionOfFeatures*. Answers the collection of features encoded by the *BitSet*.

3.1.2 Calculating dependencies

Algorithm1 represents the code for the generation of feature dependencies of *aFeatureModel.* This code is inspired by the well known Dijkstra Algorithm in order to reduce the computation complexity.

```
FeatureModel>>generateDependencies
self initializeDependencies.
encodersHaveChanged := true.
[encodersHaveChanged]whileTrue:[
encodersHaveChanged:= self updateDependencies
]
```

Algorithm1: The generation code of dependency sets.

The code starts with the initialisation of the dependencies with the neighbour features[3]. This operation is trivial from the feature model (for instance the inclusions of $feature_i$ can be initialised with:

1-The features that $feature_i$ requires according to a cross-tree constraint,

2-the parent of $feature_i$

3 The-mandatory subfeatures of $feature_i$

Then, a loop structure updates the dependencies of every feature by iteratively including the neighbours' dependencies. This loop is exited as soon as no more dependencies are found.

Algorithm2 represents the Smalltalk code we implemented to update the dependencies of every feature with those of its neighbours1.

```
FMFeature>>updateWithImplyings:
  aCollectionOfNeighbors
aCollectionOfNeighbors do:[:aFeature|
  self inclusions bitUnion: aFeature inclusions.
  feature impliedByOf bitUnion: self impliedBy.
  Self exclusions bitUnion: feature exclusionsOf
  ]
^ self hasChanged.

FMFeature>>updateWithExclusions:
  aCollectionOfNeighbors

 aCollectionOfNeighbors do:[:aFeature|
  self exclusions bitUnion: aFeature impliedByOf.
]
^ self hasChanged.
```

Algorithm2: updating features dependencies with those of the neighbours.

The update is made according to the definitions of the dependency sets given in section3. Namely, for a $feature_i$:

- if $feature_i$ implies $feature_k$ and $feature_k$ implies $feature_j$ then $feature_i$ implies $feature_j$
- if $feature_i$ implies $feature_k$ and $feature_i$ isImplyedBy $feature_j$ then $feature_k$ is implyedBy $feature_j$
- if $feature_i$ implies $feature_k$ and $feature_k$ excludes $feature_j$ then $feature_i$ excludes $feature_j$
- and finally:
- if $feature_i$ excludes $feature_k$ and $feature_k$ isImpliedBy $feature_j$ then $feature_i$ excludes $feature_j$

At the end of the overall process, we have the dependencies of every feature of the FM. Next, we propose to exploit these dependencies to efficiently implement the different operations on features models.

[3] neighbhour feature is the one that is directly dependent

3.2 Exploiting features dependencies to efficiently implement FM operations.

From the features dependencies, we can trivially define the operations 1, 6 and 8 as follows:

1. Feature model satisfiability:

A feature model is satisfiable if for all the features the intersection of the inclusions and exclusions sets is an empty collection:

```
FeatureModel>>isSatisfiable
features do:[:aFeature|
   [(aFeature inclusions bitIntersection:
                          aFeature)
         asCollectionOfFeatures ]
        ifNotEmpty:[^false]
]
^true
```

6.Dead Feature

Dead features are all the features that belong to the exclusions set of the root feature:

```
FMFeatureModel>>deadFeatures
^self root exclusions asCollectionOfFeatures
```

8. Product validation

A product consisting in a collection of (selected) features is valid if and only if for every feature, if the feature is selected in the product, all its implied features are selected and all its exclusions are deselected. If the feature is deselected, all the features that imply it are deselected too:

```
FeatureModel>>IsValidProduct: aCollecOfFeatures
Self features do:[:aFeature|
(aCollecOfFeatures contains: aFeature) ifTrue:[
((aCollectionOfFeatures containsAll: aFeature
inclusions)
        and:
aCollectionOfFeatures containsNone:
aFeature exclusions)) ifFalse:[^false]
]
ifFallse:[
(aCollectionOfFeatures containsNone:
aFeature implyedBy)
    ifFalse: [^ false]
]
^ true
```

In order to implement other operations, we introduce the *transitiveSelectionOperator*. The intuition behind the *transitiveSelectionOperator* is to instantly propagate the selection or de-selection of a given feature based on its dependencies. Algorithm3 depicts this operator.

```
FMFeature>>selectIn: aBitSet

   "aBitSet encodes a potential product"

self set: aBitSet atAll:self implyings at: 1
self set: aBitSet atAll: self exclusions at: 0.

FMFeature>>desectIn aBitSet
self set aBitSet atAll: self impliedBy at:0.
```

Algorithm.3 The Transitive Selection Operator

These two operations can be exploited to implement FM operations as follows:

2. Calculating the derivable products: This operation simply generates valid products by brute force calculation. This operation is therefore NP hard. However, using the transitive selection operator reduces the complexity of the algorithm because the search space is drastically reduced as multiple features are set simultaneously. Besides, the generated product is guaranteed to be valid Therefore, there is no need the check its validity (which is an additional factor for increasing the brute

force algorithm). To this end, we implemented the method that brute forces the solutions as follows:

```
FeatureModel>>evaluateSubconfigurationsOf:
aBitSet at: index

(aBitSet at: index) ifNil:[
 "feature at index has not been set yet"

  self deSelectandExploreFeatureAt: index in:
  aBitSet.
  selectAndExploreFeatureAt: index in aBitSet
]
ifNotNil:[
 "feature has ealready been set, then set the
next one"
  evaluateSubconfigurationsOf:aBitSet at:index+1
]
```

Where:

`selectAndExploreFeatureAt: i in: aBitSet` is a recursive method that performs a transitive selection of the $feature_i$. The result is a subset of derivable products that include $feature_i$. The method sets bit i and all dependent bits (bits corresponding to the implying and exclusions of $feature_i$) in `aBitSet`. If all the bits are set in the bitset, (we have a final product) then the product is added to the set of derivable ones else, the algorithm continuous evaluating sub configurations with the next index. The deselection operation is performed in according to the same logic with the method: `deselectAndExploreFeatureAt:i from:aBitSet`

Then, the process of finding derivable products starts with selecting the root feature (index = 1) in a bitSet of nil elements:

```
FeatureModel>>derivableProducts
self selectAndExploreFeatureAt:1 in:
nilElemBitSet.
^ products.
```

The number of derivable products is simply obtained as follows :

```
FeatureModel>>numberOfDerivableProducts
^ self derivableProducts size
}
```

3. Calculating commonality:

This is the ratio of products including a given feature to the total number of derivable products:

```
FeatureModel>>RatioOfProductsIncluding: aFeature
products:=self selectAndExploreFeatureAt: (self
indexOf:aFeature) in: nilElemBitSet.
^ (products size)/self numberOfDerivableProducts
```

4. Optimizing

Optimizing can be performed by associating fitness values to features, then evaluating all the products generated with `allDerivableProducts` and selecting the one with the highest fitness. However, proceeding this way is time consuming and thus inapplicable to large feature models (FM with many hundreds or thousands of features). To be effective, optimizing needs to use some meta-heuristics as will be shown in the next section.

5. Generating random Feature models

Exploiting the dependencies between features, it is possible to generate random feature models guarantying their validity. In order to avoid generating contradictory crosstree constraints, the generation algorithm uses the `chooseRightSideFeature` method which takes as argument the `leftSideFeature`. This is to avoid connecting two features that are already dependent in an inconsistent way. This may result in an invalid feature model if for instance the `leftSideFeature` is the parent of the `rightSideFeature` and the generated constraint is an exclusion constraint.

In the next section the genetic algorithm for optimal features selection is presented and serves to illustrate the benefit of the proposed dependence relationship.

4. Evaluating The Proposed Approach: A Genetic Algorithm for optimal features selections based on the transitive selection operator

In order to evaluate our proposal, we implemented a genetic algorithm features' selection optimization based on features' dependencies. Features' selection optimization has been recognized as a complex and time consuming task. Indeed, calculating and evaluating all the products is only possible for small feature models whereas it is impractical for large ones.

4.1 Genetic algorithms

Genetic algorithms are "stochastic-based search techniques that comprise a population of individuals, where each individual encodes a candidate solution in a chromosome" [35]. They are inspired by biological evolution of chromosomes which includes mutation, recombination, and selection [36][37]. The main idea behind genetic algorithms is to gradually evolve an initial set of (possibly random) solutions for an optimization problem, to newer ones in a way that their fitness is improved from generation to generation.

Figure 3 depict the general behaviour of genetic algorithms

The genetic algorithm starts with an initial population of

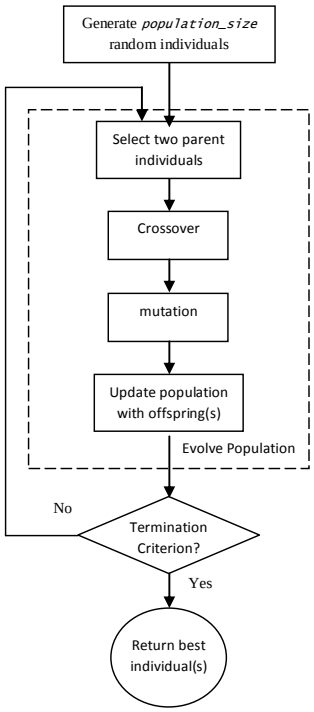


Figure 3. Genetic algorithms main steps

`population_Size` individuals. These individuals are generated randomly and evolved throughout generations. The evolution of solutions is performed through selection, crossover and mutation. This process is repeated until some condition is verified (typically a predefined number of generations is reached)

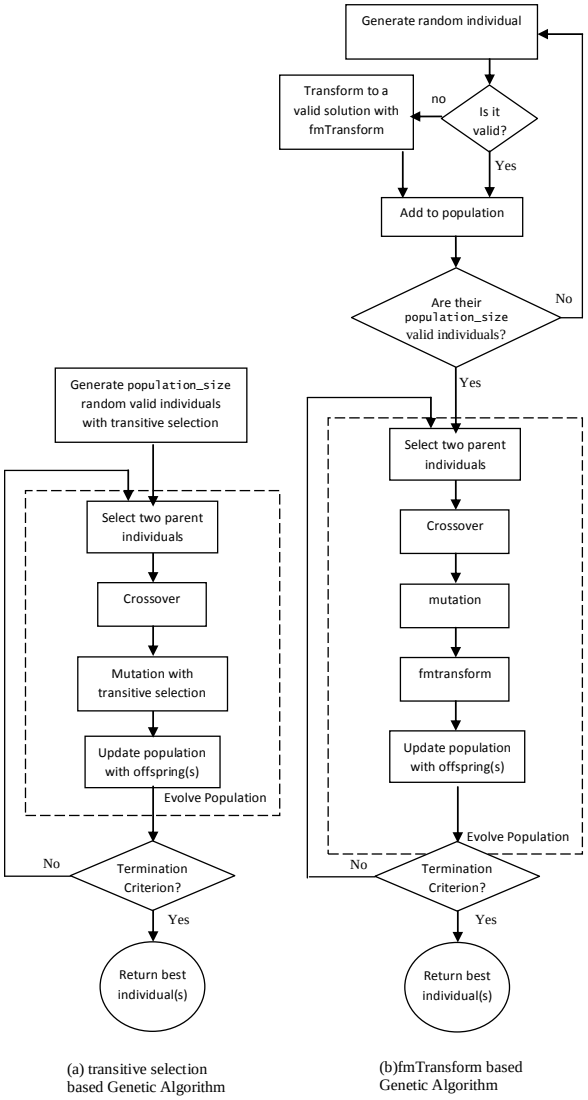
Selection consists in choosing the parent chromosomes based on their fitness. With this survival of the fittest strategy, Genetic Algorithms often quickly converge to a (near) optimal solution for optimization problems.

Crossover mixes two good parent solutions hopefully producing a better one (the offspring). For the purpose of our experimentation we implemented the uniform crossover. The uniform crossover uses a random N-sized binary mask to describe the way each offspring gene takes its value from the first parent (`mask geneAt:i = 1`) or the second parent (`mask geneAt:i = 0`)

Mutation in the other hand simply flips individuals' genes according to a specific mutation rate *M_Rate*. The role of the mutation operator is to enhance the exploration capabilities of the genetic algorithm and getting trapped in local optimums.

4.2 Applying genetic algorithms for optimal features selection

Employing genetic algorithms to optimise the selection of features for products derivation has already been explored in various previous approaches [11][8][21][39]. In this context, an individual represents a particular product i.e. a valid combination of features. To encode such individuals, binary chromosomes are used. They consist in arrays of bits where the element at index i equals 1 if the $feature_i$ is included in the encoded product and 0 otherwise.



(a) transitive selection based Genetic Algorithm

(b)fmTransform based Genetic Algorithm

Algorithm 4. Genetic algorithms for features selection

It is worth mentioning that individuals that are randomly generated by the GA operations are most likely to be invalid. In order to fix that, one can transform the generated individuals to conform the FM constraints. This approach has been implemented by Guo et al in [11] and extended in [8] and [21]. In the contrary, we proposed new initialisation and mutation operators that generate only valid individuals using features dependencies and the `transitiveSelection` operator (please refer to [40] for more details).

Algorithm 4.a depicts the general structure of the genetic algorithm based on transitive selection whereas Algorithm4.b depicts Guo's Genetic algorithm.

As mentioned earlier the genetic algorithm of Guo et al transforms randomly generated individuals so that they become valid. This is performed through the implementation of the *fmTransform* operator [11]. This operator uses two operations: *includeFeature*(f) and *excludeFeature*(f) to determine whether a feature should be included or be excluded. They are recursively implemented according to FM Constraints as follows [11]:

a. **includeFeature**(*f*)

1. if *f* is not root then **includeFeature** the parent of *f*.
2. if *f* is in an Alternative group, then **excludeFeature** all the brothers of *f* in the Alternative Group.
3. if *f* has mandatory sub features then **includeFeature** all mandatory sub feature of *f*.
4. if *f* is part of a require crosstree constraint then **includeFeature** all Features crosstree required by *f*.
5. if *f* is part of an exclude crosstree constraint then **excludeFeature** all features that *f* CrossTree Excludes
6. if *f* has any sub features then **includeFeature** at least one sub feature of *f*

b. **excludeFeature**(*f*)

1. if *f* has any sub features then **excludeFeature** all subfeatures of *f*
2. if: *f* is a mandatory SubFeature of another feature **excludeFeature** the parent of *f*
3. if a feature crosstree Requires *f* then **excludeFeature** all features that crosstree requires *f*

Obviously, the *fmTransform* operator selection and deselection operations are heavily recursive while the transitive selection operator is not. Our intuition is then that our genetic algorithm will be faster than Guo's genetic algorithm. To confirm this intuition, we performed a certain number of experimentations as explained in the next sub section.

4.3 Comparing the two genetic algorithms

Hereafter, empirical results are reported from our experiments and used to evaluate the proposed approach. The objective of these experiments is to asses about the efficiency of using features dependencies to reason about feature models. We compare our algorithm to the one presented in [11] in terms of solutions quality and convergence time to show that the first one can produces solutions of equivalent quality in less time. The last comparison criterion is the number of explored combinations in order to ensure that the use of dependencies

between features does not restrain the exploration capabilities of the genetic algorithm.

4.3.1 Experimental setup

We performed our experiments on randomly generated feature models with different sizes. To this end we adopted the method presented in [22]. According to the survey conducted by the authors, this method generates feature models reflecting the characteristic of software families that can be met in real world.

To generate optimisation problems with the randomly generated feature models, we associated random values in the range minVal, maxVal to each feature. The fitness value of a given solution is then obtained as the sum of the utility values of all active features in this solution: *objectiveFunction = Σ (utility xi) such as xi=1*. in the presence of multiple objectives, a vector of randomly generated utility values is associated to each feature and each objective function Fi equals the sum of the fitness value i of each selected feature.

We applied the two genetic to a number of feature models with varying sizes. We measured the time to generate a (near) optimal solution, the total number of generated solutions and the quality of the best found solutions. The above experiments are repeated 100 times for each generated feature model in order to avoid the fluctuations caused by random parameters.

To support the experiments we implemented all the algorithms in Pharo/Smalltalk. and performed the experiments on an 2,66 GHz Intel Core 2 Duo MacBook Pro running the Mac OS X Yosemite and having 8GB of 1067 MHz Ram.

4.4 Discussion of results and threats of validity

We have executed our genetic algorithm and compared its performances with the genetic algorithm presented in [11]. We have configured our genetic algorithm as well as well the genetic algorithm of [11] with the following parameters:

Population size: 800
Mutation Rate: 0.1
Stop criteria: number of generation < 800
Uniform Rate: 0.5

We compared the different algorithms according to tree values:

a) The of number of explored solutions : this is to assess about the exploration capabilities of the genetic algorithms

b) The fitness of the best individuals : this is to compare about the convergence capabilities of the genetic algorithms

c) The total time: this is to compare the response time of the different solutions.

Table 1 shows the experimental results obtained from the comparison of different runs of the two genetic algorithms presented above on feature models of varying runs. The first column of Table 1 shows the size (i.e. number of features) of the feature models. Columns 2 and 3 show the execution times of our genetic algorithm and the one from [11] respectively, whereas column 4 shows the ratio of the execution time of the first to the second. Column 5 shows the ratio of the number of explored combinations of both algorithms and column 6 the ratio of their best fatnesses. We have repeated each experience 100 times and calculated the medium of each column.

From Table 1, one can clearly notice that our genetic algorithm evolves more quickly than the one from [11] without any significant impact on performances. Indeed, the ratios of best fitness values as well as the ratio of the number of explored combinations oscillate around 1 and the medium variation value is less than 0.01.

In the other hand, it is obvious from Table 1 that the genetic algorithm exploiting features dependencies converges in less time than the one from [11]. Moreover, the difference between the two execution times is proportional to the number of feature models and our algorithm can be 99% faster than the one from the literature.

Table 1. The experimental results

Number Of Features	Execution time of our GA (in seconds)	Execution Time of the GA from [11] (in seconds)	Gain in execution time	Ratio of the number of explored combinaisons	Ratio of the best fitness
100	1,00	8,00	88%	1,059820576	0,944061692
200	2,00	32,00	94%	0,999064291	1,042535895
500	4,00	165,00	98%	1,030866725	1,032808419
800	14,00	634,00	98%	0,999769554	0,946222716
1000	15,00	682,00	98%	1	0,904538341
2000	26,00	2056,00	99%	1	0,843161857

5. RELATED WORKS

Several works have addressed the problem of automatic reasoning on feature models and panoply of approaches and tools exist to support it. A good survey on the topic can be found for instance in [19] and [20].

A first category of approaches uses propositional logic to analyse FMs. For instance, in [23], authors propose to translate FMs into propositional formulas and some operations were implemented accordingly. Zhang et al [24] and Sun et al [25] formalised FM using SVM and Z + Alloy Analyser respectively and were able to formulate some basic operations such as dead features detection and explanation in case of FM inconsistency. In [38] authors suggest the use of SAT solvers and constraint propagation algorithms to the same purpose.

Another family of techniques uses description logic reasoners [26]. A typical example is the translation of FMs into a sublanguage of OWL and the exploitation of tools such as the Renamed ABox and Concept Expression Reasoner to perform certain operations on FMs.

CSP Solvers have also been used in this perspective [27] [28]. To this end, the FM is translated into a Constraint Satisfaction Problem. Then, constraint programming techniques are leveraged to support automatic operations not only on classical feature models but also on some extended ones.

Finally, in [29] Cao et al. propose to transform FMs to a normalised form then encode the FM in a particular data structures that represents the valid combinations of the parent feature and its children. The children themselves can be encoded as sub features models. The proposed approach does not seem to deal with cross-tree constraints or support future integration of extended forms of FMs.

6. CONCLUSION AND PERSPECTIVES

Software Product Lines engineering is a promising paradigm to optimize core assets reuse in order to enhance software quality, costs and time to market. At the heart of SPL engineering is variability modelling. In this sense variability models (and more specifically, feature models) not only serve to help developers identify the reusable assets of the software factory, but also help them detect design inconsistencies and help stakeholders making the right decisions.

From this perspective, the automatic reasoning on feature models is a critic task. Often, as the size of modern systems

increases, this operation can be very time consuming. This even more true in the case of Dynamic SPL where the automatic operations are performed on runtime.

In the present article we introduced a new concept of features dependencies. This notion has then been exploited to efficiently implement some operations on FMs. To assess the efficiency of our proposal, we implemented a genetic algorithm for features optimisation and compared it to one notable one from the literature. The experimental results clearly show the superiority of the dependencies based genetic algorithm in terms of convergence time while quality of generated solutions is maintained.

The article represents a first evaluation of the notion of dependencies. Comparison with other existing approaches (especially, SAT and CSP based approaches) needs to be performed in the future. The generalisation of features dependencies to extended feature models is also an interesting perspective (For instance, feature attributes can be handled as sub-features as it has been suggested in [8] for instance). Lastly, the performance of FM operations can be enhanced by changing the BitSet encoding as arrays of bits instead of arrays of integers as operations on bits can be performed much quicker.

Acknowledgments

This work has been partially supported by ESUG, the European Smalltalk Users Group. The hospitality offerd by Dr S. Ducasse of the Rmod team at Lille is gratefully acknowledged. The authors would also like to thank Dr. N. Anquetil for proofreading and helpful comments.

References

[1] F. van der Linden, K. Schmid, and E. Rommes. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[2] IEEE Std 829-1998. "IEEE Standard for Software Test Documentation". September 16th 1998.

[3] S. Apel and C. K¨astner. An overview of feature-oriented software development. Journal of Object Technology (JOT), 8(5):49–84, 2009.

[4] Mark Dalgarno. Software Product Line Engineering with Feature Models, Design of applications and programs, Overload Journal #78 - Apr 2007

[5] K. Kang, et al., Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990

[6] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000

[7] klaus pohl, gruner bockle, and frank j. van der linden Software product line engineering : foundation, principles and techniques. Springer_verlag 2005

[8] Gustavo G. Pascual, Mónica Pinto, Lidia Fuentes, Self-adaptation of mobile systems driven by the Common Variability Language, Future Generation Computer Systems, Volume 47, June 2015, Pages 127-144, ISSN 0167-739X, http://dx.doi.org/10.1016/j.future.2014.08.015.

[9] M. Harman, Y. Jia, J. Krinke, B. Langdon, J. Petke, Y. Zhang, Search based software engineering for software product line engineering: a survey and directions for future work (keynote paper), in: 18th International Software Product Line Conference (SPLC 14), Florence, Italy, 2014, pp. 5–18.

[10] A. Sayyad, J. Ingram, T. Menzies, H. Ammar, Scalable product line configuration: A straw to break the camel's back, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 465–474.

[11] J.Guo,J.White,G.Wang,J.Li,Y.Wang,A genetic algorithm for optimised feature selection with resource constraints in software product lines, Journal of Systems and Software 84 (12) (2011) 2208 – 2221.

[12] S. Soltani, M. Asadi, D. Gasevic, M. Hatala, E. Bagheri, Automated planning for feature model configuration based on functional and non-functional requirements, in: Proceedings of the 16th International Software Product Line Conference-Volume 1, ACM, 2012, pp56–65.

[13] J. White, B. Dougherty, D. C. Schmidt, Selecting highly optimal architectural feature sets with filtered cartesian flattening, Journal of Systems and Software 82 (8) (2009) 1268 – 1284.

[14] J. White, B. Dougherty, D. Schmidt, D. Benavides, Automated reasoning for multi-step feature model configuration problems, in: Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University, 2009, pp. 11–20.

[15] J.White,D.Schmidt,E. Wuchner,A. Nechypurenko,Automating productline variant selection for mobile devices, in: Software Product Line Conference, 2007. SPLC 2007. 11th International, IEEE, 2007, pp. 129–140. [16] G. Brataas, S.O. Hallsteinsen, R. Rouvoy, F. Eliassen, Scalability of decision models for dynamic product lines in: SPLC, (2), 2007, pp. 23–32.

[17] André Almeida, Francisco Dantas, Everton Cavalcante, Thais Batista. A Branch-and-Bound Algorithm for Autonomic Adaptation of Multi-Cloud Applications. IEEE. 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014), May 2014, Chicago, United States. pp.315-323.

[18] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines.

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

[20] D. Benavides, P. Trinidad, A. Ruiz-Cortes, Automated reasoning on feature models, in: Advanced Information Systems Engineering, Springer, 2005, pp. 381–390.

[21] Gustavo G. Pascual, Roberto E. Lopez-Herrejon, Monica Pinto, Lidia Fuentes, Alexander Egyed, Applying Multiobjective Evolutionary Algorithms to Dynamic Software Product Lines for Reconfiguring Mobile Applications, The Journal of Systems & Software (2015), doi: 10.1016/j.jss.2014.12.041

[22] Thum, T., Batory, D. S., Kastner, C., 2009. Reasoning about edits to feature models. In: Proceedings of ICSE'09, Van- couver, Canada, pp. 254-264.

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

.[24] W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. In J. Davies, editor, ICFEM 2004, volume 3308, pages 115–130. Springer–Verlag, 2004.

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

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

[27] D. Benavides, A. Ruiz-Cort´es, B. Smith, Barry O'Sullivan, and P. Trinidad. Computational issues on the automated analyses of feature models using constraint programming. International Journal

of Software Engineering and Knowl- edge Engineering, in preparation, 2006.

[28] D. Benavides, A. Ruiz-Cort´es, and P. Trinidad. Using constraint programming to reason on feature models. In The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, 2005.

[29] F. Cao, B. Bryant, C. Burt, Z. Huang, R. Raje, A. Olson, and M. Auguston. Automating feature-oriented domain analysis. In International Conference on Software Engineering Research and Practice (SERP'03), pages 944–949, June 2003.

[30] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano, "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes", IEEE Computer, 2009, pp 46-52.

[31] A. S. Karata¸s, H. Oguztüzün, and A. Dogru. "Global Constraints on Feature Models". Proceedings of Principles and Practice of Constraint Programming - 16th International Conference (CP-2010), Scotland 2010. Springer, vol. 6308, pp. 537-551. ISBN 9783642153952.

[32] Carlos Eduardo Alvarez Divo, Automated Reasoning on Feature Models via Constraint Programming, master thesis., June 2011

[33] A. S. Karata¸s, H. Oguztüzün, and A. Dogru. "Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains". Proceedings of Software Product Lines: Going Beyond - 14th International Conference, (SPLC-2010), South Korea 2010. Springer, vol. 6287, pp. 286-299. ISBN 9783642155789.

[34] Nelly Bencomo, Jaejoon Lee, Svein O. Hallsteinsen, "How dynamic is your Dynamic Software Product Line?". In proceeding of: Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings

[35] J. H. Holland. Adaptation in Natural and Artificial Systems. MIT Press, Cambridge, MA, USA, 1992

[36] Faezeh Ensan, Ebrahim Bagheri, Dragan Gasevic: Evolutionary Search-Based Test Generation for Software Product Line Feature Models. CAiSE 2012: 613-628

[37] K. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Doctoral Dissertation. Ann Arbor: The University of Michigan, 1975.

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

[39] A Alidra, MT Kimour. « A new evolutionary approach to decision-making in autonomic systems ». In Proceedings of the 3th IEEE control International Conference on System and control (), Algiers 2013

[40] A. Alidra, M.T. Kimour, A new genetic algorithm for enhanced decision-making in large adaptable systems" submitted to the International Journal of Computational Science and Engineering (IJCSE).

[41] S. Urli, A. Bergel,M, Blay-Fornarino, P. Collet. A visual support for decomposing complex feature models IEEE 3rd Working Conference on Software Visualization. Pages 76 - 85 Bremen. Sept. 2015