Automated Reasoning on Feature Models [*]

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés

Dpto. de Lenguajes y Sistemas Informáticos
University of Seville
Av. de la Reina Mercedes S/N, 41012 Seville, Spain
{benavides, trinidad, aruiz}@tdg.lsi.us.es

Abstract Software Product Line (SPL) Engineering has proved to be an effective method for software production. However, in the SPL community it is well recognized that variability in SPLs is increasing by the thousands. Hence, an automatic support is needed to deal with variability in SPL. Most of the current proposals for automatic reasoning on SPL are not devised to cope with extra–functional features. In this paper we introduce a proposal to model and reason on an SPL using constraint programming. We take into account functional and extra–functional features, improve current proposals and present a running, yet feasible implementation.

Keywords: Product Lines, Feature Models, Extra–functional Features, Automated Reasoning.

1 Introduction and Motivation

Research on SPLs is thriving. Unlike other approaches reuse in SPL has to become systematic instead of ad–hoc. In order to achieve such a goal, SPL practices guide organizations towards the development of products from existing assets rather than the development of separated products one by one from scratch. Thus, features that are shared by all SPL products are reused in every single product. Most of the existing methods [3,6] for SPL engineering agree that a way for modelling SPL is needed. In this context feature models [7,9,11,13,22] have been quoted as one of the most important contributions to SPL modelling [7, pag.82]. As in other cases, first applications in routine production are stimulating the development of a supporting science for improving the production methods [17].

Feature models are used to model SPL in terms of features and relations amongst them. In this type of models, the number of potential products of an SPL increases with the number of features. Consequently, a large number of features lead to SPLs with a large number of potential products. In an extremely flexible SPL, where all features may or may not appear in all potential products, the number of potential products is equal to 2^n, being n the number of features. Moreover, current feature models are only focused on modelling functional features and in the most quoted proposals [7,9,11,13,22] there is a lack of modelling artifacts that deal with extra–functional features (features related

to so–called quality or non–functional features). If extra–functional features are taken into account the number of potential products increases even further. Although it is accepted that in an SPL it is necessary to deal with these extra–functional features [5,11,12], there is no consensus about how to deal with them.

Automated reasoning is an ever challenging field in SPL engineering [18,23]. It should be considered specially when the number of features increases due to the increase in the number of potential products. To the best of our knowledge, there are only a couple of limited attempts by Van Deursen *et al.* and Mannion [8,14] that treat automatic manipulation of feature models. Although those proposals only consider functional features, leaving out extra–functional features. Van Deursen *et al.* [8] explore automated manipulation of feature descriptions providing an algebra to operate on the feature diagrams proposed in [7]. Mannion's proposal [14] uses first–order logic for product line reasoning. However it only provides a model based on propositional–logic using AND, OR and XOR logical operators to model SPLs. Both attempts have several limitations:

1. They do not allow to deal with extra–functional features (both attempts leave this work pending).
2. They basically answer to the single question of how many products a model has.
3. As far as we know, they have no available an implementation.

In addition, Mannion's model uses the XOR ($\oplus$) operator to model alternative relations, which is either a mistake or a limitation because the model becomes invalid if more than two features are involved in an alternative relation.

The contribution of this paper is threefold. First, we extend existing feature models to deal with extra–functional features. Secondly, we deal with automatic reasoning on extended feature models answering five generic questions, namely i) how many potential products a model has ii) which is the resulting model after applying a filter (e.g. users constraint) to a model, iii) which are the products of a model, iv) is it a valid model, and v) which is the best product of a model according to a criterion and finally giving an accessible, running implementation.

The remainder of this paper is structured as follows. In Section 2, we propose an extension to deal with extra–functional features. In Section 3, we present a mapping to transform an extended feature model into a Constraint Satisfaction Problem (CSP) in order to formalize extended feature models using constraint programming [15]. In Section 4, we improve current reasoning on feature models and we give some definitions to be able to automatically answer several questions on extended feature models. In Section 5, we show how our model can be applied to other important activities such as obtaining commonality and variability information. In Section 6, we briefly present a running prototype implementation. Finally, we summarize our conclusions and describe our future work in Section 7.

2 Extending Feature Models with Extra–Functional Features

2.1 Feature Models

The main goal of feature modelling is to identify commonalities and differences among all products of a SPL. The output of this activity is a compact representation of all potential products of an SPL, hereinafter called *"feature model"*. Feature models are used to model SPL in terms of features and relations among them. Roughly speaking, a feature is a distinctive characteristic of a product. Depending on the stage of development, it may refer to a requirement [10], a component in an architecture [2] or even to pieces of code [16] (feature oriented programming) of a SPL.

There are several notations to design feature models [7,9,11,13,22]. We found that the one proposed by Czarnecki is the most comprehensible and flexible as well as being one of the most quoted. Figure 1 depicts a possible feature model of an SPL for the domain of Home Integration Systems (HIS) using Czarneky's notation. This example is partially inspired by [13].
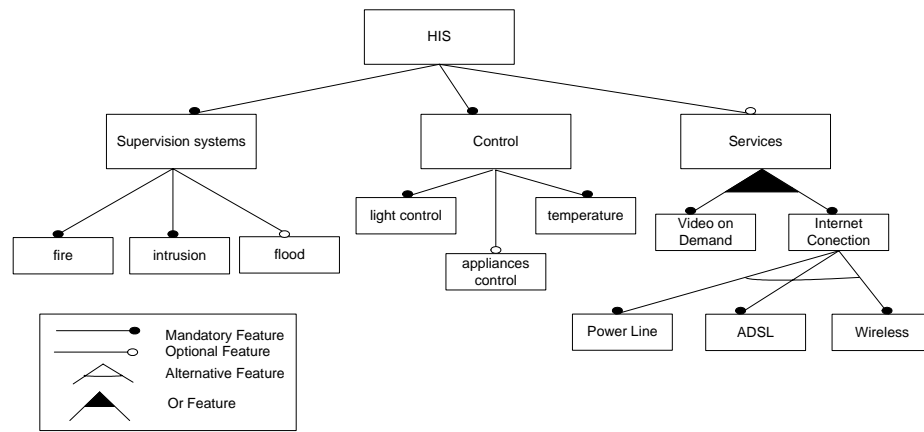
Figure 1. Feature model of an SPL in the HIS domain

Czarnecki's notation proposes four relations, namely: mandatory, optional, alternative and or–relation. In these relations, there is always a parent feature and one (in the case of mandatory and optional relations) or more (in the case of alternative and or–relation) child features.

– **Mandatory**: the child feature in this relation is always present in the SPL's products when its parent feature is present. For example, Every HIS is equipped with $i)$ fire and intrusion supervision systems and $ii)$ light and temperature control.
– **Optional**: the child feature in an optional relation may or may not be present in a product when its parent feature is present. For Example, there are HISs with services and others without them.
– **Alternative**: a child feature in an alternative relation may be present in a product if its parent feature is included. In this case, only one feature of the set of children is present. For example, in a HIS product if an Internet connection is included, then

the customer has to choose between an ADSL, powerline or wireless connection, but only one.

- **Or–relation**: the child feature in an or–relation may be present in a product if its parent feature is included. Then, at least one feature of the set of children may be present. For example, in a HIS the products may have Video or Internet or both at the same time.

This model includes 32 potential products (you can check this on *http://www.tdg-seville.info/topics/spl*). Examples of them are: i)Basic product: consisting of a fire and intrusion supervision systems and light and temperature control. ii)Full product: a product with all supervision and control features as well as a power line, ADSL or wireless Internet connection.

2.2 Extended Feature Models

Current proposals only deal with characteristics related to the functionality offered by an SPL (functional features). Thus, there exists no solid proposal for dealing with the remaining characteristics, also called extra-functional features. There are several concepts that we would like to clarify before analyzing current proposals and framing our contribution:

- Feature: a prominent characteristic of a product. Depending on the stage of development, it may refer to a requirement [10] (if products are requirement documents), a component in an architecture [2] (if products are component architectures) or even to pieces of code [16] (if products are binary code in a feature oriented programming approach) of an SPL.
- Attribute: the attribute of a feature is any characteristic of a feature that can be measured. *Availability* and *cost* are examples of attributes of the *Service* feature of figure 1. *Latency* and *bandwidth* may be examples of attributes of an Internet connection.
- Attribute domain: the space of possible values where the attribute takes its values. Every attribute belongs to a domain. It is possible to have discrete domains (e.g:integers, booleans, enumerated) or continuous domains (e.g.:real).
- Extra–functional feature: a relation between one or more attributes of a feature. For instance: $bandwidth = 256, Latency/Availability > 50$ and so on. These relations are associated to a feature.

In figure 1, every feature refers to functional features of the HIS product line so that every product differs because of its functional features. However, every feature of figure 1 may have associated extra–functional features. For instance, considering the *services* feature, it is possible to identify extra–functional features related to it, such as relations among attributes like *availability,reliability, development time, cost* and so forth. Likewise the *Internet Connection* feature can have extra–functional features such as relations among *latency* or *bandwidth*. Furthermore, the attributes' values of extra–functional features can differ from one product to another. It means, every product not only differs because of its functional features, but because of its extra–functional features too.

Consider the full product of the HIS product line example presented formerly with the same functional features. It is possible to offer several products with the same functional features but different extra–functional features, for instance: i) High quality full product: a product with full functionality and high quality: high *availability* and *reliability* and high *cost* too. ii) Basic quality full product: a product with full functionality but lower quality: lower *availability* and *reliability* and lower *cost* too.

To date, we have not found any proposal dealing with functional and extra–functional features in the same model. However, there are some works in the literature that suggest the need of dealing with extra–functional features: Kang *et. al* have been suggesting the need to take into account extra–functional features since 1990 [11, pag. 38] when they depicted a classification of features, although they did not provide a way to do it. Later, in 1998 Kang *et. al* [12] made an explicit reference to what they called 'non–functional' features (a possible type of what we call extra–functional features). However the authors still did not propose a way to solve it. In 2001 Kang *et. al* [5], proposed some guidelines for feature modelling: in [5, pag. 19], the authors once again made the distinction between functional and quality features and pointed out the need of a specific method to include extra–functional features, but they did not provide this specific method on this occasion either.

2.3 A Notation for Extended Feature Models

We propose to extend Czarneki's feature models with extra–functional features and improve previous vague notations proposed in [20] by allowing relations amongst attributes. Using the HIS example, every feature may have one or more attribute relations, for example, the *price* ($PRICE$) and *development time* (expressed in hours) ($DTIME$) taking a range of values in both a discrete or continuous domain (integer or real for example). Thus, it would be possible to decorate the graphical feature model with this kind of information. Figure 2 illustrates a piece of the feature model of figure 1 with extra–functional features with our own notation inspired by [20].
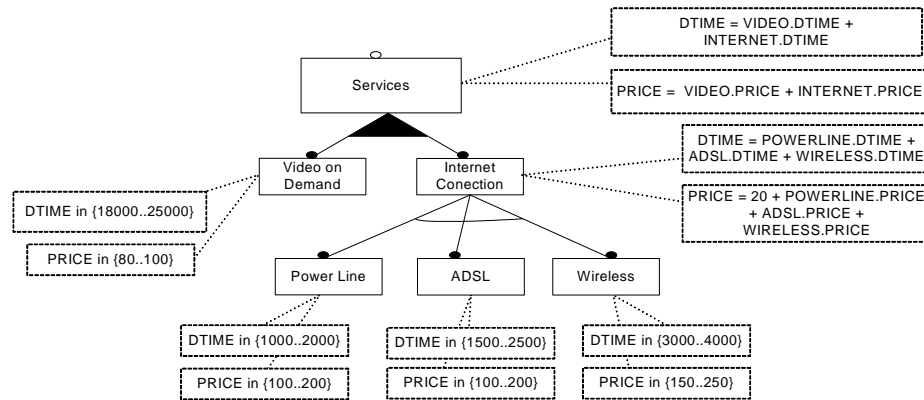

Figure 2. Extended feature model for an SPL in the HIS domain

In this example, every sub feature of the *Service* feature has two attributes: $PRICE$ and $DTIME$. Each of the attributes of leaf features are in a domain of values. For instance, the price of an ADSL connection can range from 100 to 200 [1]. In the case of parent features, the values of the attributes are the addition of their children values. For example, the price of an Internet connection is the sum of the prices of the possible Internet connections.

3 Mapping Extended Feature Models onto CSP

3.1 Preliminaries

Constraint Satisfaction Problems [21] have been object of research in Artificial Intelligence in the last few decades. A Constraint Satisfaction Problem (CSP) is defined as a set of variables, each ranging on a finite domain, and a set of constraints restricting all the values that variables can take simultaneously. A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied simultaneously. We may want to find: $i)$ just one solution, with no preference as to which one, $ii)$ all solutions, $iii)$ an optimal solution by means of an objective function defined in terms of one or more variables. Solutions to a CSP can be found by searching (systematically) through all possible value assignments to variables.

In many real-life applications, we do not want to find any solution but a good solution. The quality of a solution is usually measured by an application dependent function called objective function. The goal is to find a solution that satisfies all the constraints and minimize or maximize the objective function, respectively. Such problems are referred to as Constraint Satisfaction Optimization Problems (CSOP), which consist of a standard CSP and an optimization function that maps every solution (complete labelling of variables) to a numerical value. These are some basic definitions of what a CSP is.

Definition 1 (CSP). *A CSP is a three–tuple of the form (V, D, C) where $V \neq \emptyset$ is a finite set of variables, $D \neq \emptyset$ is a finite set of domains (one for each variable) and C is a constraint defined on V.*

Consider, for instance, the CSP: $(\{a, b\}, \{[0..2], [0..2]\}, \{a + b < 4\})$

Definition 2 (Solution). *Let ψ be a CSP, a solution of ψ is whatever valid assignment of all elements in V as satisfies C.*

In the previous example, a possible solution is $(2, 0)$ since it verifies that $2 + 0 < 4$.

Definition 3 (Solution space). *Let ψ be a CSP of the form (V, D, C), its solution space denoted as $sol(\psi)$ is made up of all its possible solutions. A CSP is satisfiable if its solution space is not empty.*

$$sol(\psi) = \{S \mid \forall s_i \cdot s_i \in S \Rightarrow C(s_i) = true\}$$

---

[1] these values are just illustrative, they may have nothing to do with real values

In the previous example, there are eight solutions. The only assignment that does not satisfy $a + b < 4$ is $(2, 2)$. Nevertheless, if we replace the constraint with $a + b < -1$, then the CSP is not satisfiable.

Definition 4 (CSOP). *A CSOP is a four–tuple of the form (V, D, C, O) where V, D and C stand for a CSP and O is a real function defined on D.*

Consider, for instance, the CSOP: $(\{a, b\}, \{[0..2], [0..2]\}, \{a + b < 4\}, a)$

Definition 5 (Optimum space). *Let ψ be a CSOP of the form (V, D, C, O), its optimum space denoted as $max/min(\psi, O)$ is made up of all solutions that maximize or minimize O.*

$$max(\psi, O) = \{s \mid \forall s\prime \cdot s\prime \in sol(\psi) \;\wedge\; s\prime \neq s \;\Rightarrow O(s) \geq O(s\prime)\}$$

$$min(\psi, O) = \{s \mid \forall s\prime \cdot s\prime \in sol(\psi) \;\wedge\; s\prime \neq s \;\Rightarrow O(s) \leq O(s\prime)\}$$

In the previous example, $max(\psi, a) = \{(2, 0), (2, 1)\}$.

3.2 The Mapping

In [1] we presented an algorithm to transform an extended feature model into a CSP. The mapping between a feature model and a CSP has the following general form: $i)$ the features make up the set of variables, $ii)$ the domain of each variable is the same: $\{true, false\}$, $iii)$ extra–functional features are expressed as constraints and $iv)$ every relation of the feature model becomes a constraint among its features in the following way:

– Mandatory relation: Let f be the parent and f_1 the child in a mandatory relation , then the equivalent constraint is: $f_1 = f$
– Optional relation: Let f be the parent and f_1 the child in an optional relation, then the equivalent constraint is: $f_1 \Rightarrow f$
– Or–relation: Let f be the parent in an or–relation and $f_i \mid i \in [1 \ldots n]$ the set of children, then the equivalent constraint is: $f_1 \vee f_2 \vee \ldots f_n \Leftrightarrow f$.
– Alternative relation: Let f be the parent of an alternative relation and and $f_i \mid i \in [1 \ldots n]$ the set of children, then the equivalent constraint is:
$(f_1 \Leftrightarrow (\neg f_2 \wedge \ldots \wedge \neg f_n \wedge f)) \wedge (f_2 \Leftrightarrow (\neg f_1 \wedge \neg f_3 \ldots \wedge \neg f_n \wedge f)) \wedge$
$(f_n \Leftrightarrow (\neg f_1 \wedge \ldots \wedge \neg f_{n-1} \wedge f))$

There may be several different algorithms to map extended feature models. The one presented in [1] is a possible one. Hereinafter, we refer to the equivalent CSP resulting from the mapping as ψ_M. Using this mapping, constraints for functional and extra–functional features can be handled together. Thus, table 1 shows the equivalent constraints for figure 1 with the extra–functional features of figure 2. Constraints of extra–functional features are denoted by an asterisk. $POWERLINE, ADSL$ and $WIRELESS$ extra–functional features are not shown for lack of space as they are very similar to the $VIDEO.$ ones.

Relation	ψ_{HIS}
HIS 1	$(SUPERVISION = HIS)$
HIS 2	$(CONTROL = HIS)$
HIS 3	$(SERVICES \Rightarrow HIS)$
SUPERVISION 1	$(FIRE = SUPERVISION)$
SUPERVISION 2	$(INTRUSION = SUPERVISION)$
SUPERVISION 3	$(FLOOD \Rightarrow SUPERVISION)$
CONTROL 1	$(LIGHT = CONTROL)$
CONTROL 2	$(APPLIANCE \Rightarrow CONTROL)$
CONTROL 3	$(TEMPERATURE = CONTROL)$
SERVICES 1	$((VIDEO \vee INTERNET) \Leftrightarrow SERVICES)$
SERVICES *	$(SERVICES.PRICE = VIDEO.PRICE + INTERNET.PRICE) \wedge$ $(SERVICES.DTIME = VIDEO.DTIME + INTERNET.DTIME)$
VIDEO *	$((VIDEO.PRICE \in [80\,100]) \Leftrightarrow VIDEO) \wedge$ $((VIDEO.PRICE = 0) \Leftrightarrow \neg VIDEO) \wedge$ $((VIDEO.DTIME \in [18000, 25000]) \Leftrightarrow VIDEO) \wedge$ $((VIDEO.DTIME = 0) \Leftrightarrow \neg VIDEO)$
INTERNET 1	$(POWERLINE \Leftrightarrow (\neg ADSL \wedge \neg WIRELESS \wedge INTERNET)) \wedge$ $(ADSL \Leftrightarrow (\neg POWERLINE \wedge \neg WIRELESS \wedge INTERNET)) \wedge$ $(WIRELESS \Leftrightarrow (\neg POWERLINE \wedge \neg ADSL \wedge INTERNET))$
INTERNET *	$((INTERNET.PRICE = ADSL.PRICE + WIRELESS.PRICE$ $+POWERLINE.PRICE + 20) \Leftrightarrow INTERNET) \wedge$ $((INTERNET.PRICE = 0) \Leftrightarrow \neg INTERNET) \wedge$ $((INTERNET.DTIME = ADSL.DTIME + WIRELESS.DTIME$ $+POWERLINE.DTIME) \Leftrightarrow INTERNET) \wedge$ $((INTERNET.DTIME = 0) \Leftrightarrow \neg INTERNET)$

Table 1. A trace of the algorithm presented in [1] for HIS example

4 Automated Reasoning on Extended Feature Models

Since we go toward automated reasoning on feature models, a formal model of SPL becomes necessary. We propose to use Constraint Programming to reason on extended features models.

Our model is able to answer the following questions:

4.1 Number of products

One of the questions to be answered is how many potential products a FM contains. This is a key question in SPL engineering because if the number of products increases the SPL becomes more flexible as well as more complex.

Definition 6 (Cardinal). *Let M be an extended feature model, the number of potential products of M, hereinafter cardinal, is equal to the solution number of its equivalent CSP ψ_M.*

$$cardinal\,(M) = |sol\,(\psi_M)|$$

In the HIS example of figure 1 $cardinal(HIS) = 32$, simply by adding for example a new service like *Radio Streaming*, the number of potential products raises to 64. Likewise adding the attributes of figure 2 $cardinal(HIS) = 260$

4.2 Filter

There should be a way to apply filters to the model. These filters can be imposed by the users. A filter acts as a limitation for the potential products of the model. A typical application of this operation occurs when customers are looking for a product with a specific set of characteristics, that is, they are not interested in all potential products but in some of them only (those passing the filter).

Definition 7 (Filter). *Let M be an extended feature model and F a constraint representing a filter, the filtered model of ψ_M, hereinafter filter, is equal to ψ_M adding the constraint F.*

$$filter(M, F) = (\psi_M \wedge F)$$

A possible filter for the HIS example would be to ask for all products with video on demand, making the number of potential products decrease from 32 to 16. It is also possible to apply filters to attributes. For example, it would be possible to ask for all products whose prices are lower than 200, 12then

$$cardinal(filter(HIS, SERVICES.PRICE < 200)) = 44$$
(when any filter is imposed, it decreases from 260 to 44).

4.3 Products

Once ψ_M is defined, there should be a way to get the solutions of the model, that is the products of ψ_M.

Definition 8 (Products). *Let M be an extended feature model, the potential products of the model M, hereinafter products, is equal to the solutions of the equivalent CSP ψ_M.*

$$products(M) = \{s \in sol(\psi_M)\}$$

In the HIS example we would want to get all the possible products of the model or even apply a filter and then get the products. Thus $M = filter(\text{HIS,VIDEO} = \text{true})$ and $products(M) = \{s \in sol(\psi_{HIS} \wedge \text{VIDEO=true}))\}$.

4.4 Validation

A valid extended feature model is a model where at least one product can be selected. That is, a model where ψ_M has at least one solution.

Definition 9 (Valid model). *A feature model M is valid if its equivalent CSP is satisfiable.*

$$valid(M) \iff products(M) \neq \emptyset$$

The HIS model of the example is valid, but there might be situations where the constraints are not satisfiable, making the model invalid. For instance, if the Service's price is lower than 100, and a filter is imposed to have $INTERNET$, then the model is not valid:

$$valid(filter(HIS, INTERNET = \text{true} \land SERVICE.PRICE < 100)) = false$$

4.5 Optimum products

Finding out the best products according to a determinate criterion is an essential task in our model.

Definition 10 (Optimum). *Let M be an extended feature model and O an objective function, then the optimum set of products, hereinafter max and min, is equal to the optimum space of ψ_M.*

$$max(M, O) = max(\psi_M, O)$$
$$min(M, O) = min(\psi_M, O)$$

It is also possible to apply a filter to the HIS example and then ask for an optimal product. Thus, a possible optimum criterion for the HIS example would be to ask for all products with video on demand, and the minimum value for the multiplication of price and development time. In this case selected products P_{opt} are:

$$M = filter(\,HIS, VIDEO = true\,)$$
$$O = SERVICE.PRICE * SERVICE.DTIME$$
$$P_{opt} = min(M, O)$$

The model presented in this section can support current feature models. The only difference is that current feature models do not support extra–functional features which means that when using our model to reason on current feature models, attributes are not taken into account. Thus, the algorithm presented in [1] and all previous definitions remain valid for current feature models.

5 Realising the Benefits

Compared to others, our approach is very flexible because it is so easy to extend. Below, we show two more definitions based on the previous ones to demonstrate how our approach can be extended and give valuable information to SPL engineers.

5.1 Variability

As mentioned previously, feature models are composed of a set of features and relations among them. If relations restrict the number of products to only one, we are considering the lowest variability while a feature model defining no possible product would be

considered a non-valid model. On the other hand, considering no relations, the number of products within the feature model would be the highest. This case would represent the highest variability. Relations restrict the number of potential products, so variability depends on relation types.

Let a leaf feature be a feature that has no child feature. Parent features add no variability to the model, because they are feature aggregates. We define the variability factor as follows.

Definition 11 (Variability Factor(VF)). *Let M be an extended feature model, and ψ_M the equivalent CSP. Let M^V be another extended feature model, considering the leaf features in M and no relation among its features, and ψ_M^V the equivalent CSP.*

$$VF(M) = \frac{cardinal(M)}{cardinal(M^v)} = \frac{|sol(\psi_M)|}{|sol(\psi_M^V)|}$$

The variability factor in the real domain would take values ranging from 0 to 1.

VF can assist decision making. For instance, when many products are going to be developed one of the first decisions to be taken, is whether the SPL approach or traditional approach is going to be applied. A high VF may suggest an SPL approach; a low VF may suggest a traditional approach.

5.2 Commonality

In a feature model, some features will appear in every product, some in only one product and others in some products. When deciding the order in which features are going to be developed, it is very important to know which are the most common features in order to prioritize their building. Obtaining commonality information from the feature model can be feasible by asking questions to our model. We define the feature commonality as the percentage of products containing that feature.

Definition 12 (Commonality). *Let M be an extended feature model and F the feature we want to know its commonality.*

$$commonality(M, F) = \frac{cardinal(filter(M, F = true))}{cardinal(M)}$$

6 Implementation

We have already implemented some of the ideas presented in this paper using OPL Studio, a commercial CSP solver. This implementation is available at *http://www.tdg-seville.info/topics/spl*.

Three modules have been developed in our implementation: first, a feature markup language and XML Schema were agreed on. This language allows to represent the Czarnecki's feature model [7]. Secondly, a parser to transform this XML documents to a CSP following the algorithm described in [1] was developed. Finally, a web–based prototyping interface was made available to allow to test some of the capabilities of

the model. In order to test our implementation, we have modeled four problems (two academical and two real product lines) that are available on the web site.

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java. We ran our tests on a WINDOWS XP PROFESSIONAL machine that was equipped with a 1.5Ghz AMD Athlon XP microprocessor, and 496 MB of DDR 266Mhz RAM memory. The test was based on the feature model in Figure 1, adding new features. Several tests were made on each feature model in order to avoid as many exogenous interferences as possible.

We have experimentally inferred that the implementation presented has an exponential behavior while increasing the number of features in the feature model and maintaining a constant variability factor. We have measured the solving time for $products(M)$, which is the most complex to obtain, and have considered it for different values of VF as shown in Figure 3. Our test determines our model has a good performance up to 25 features while the VF is kept constant.
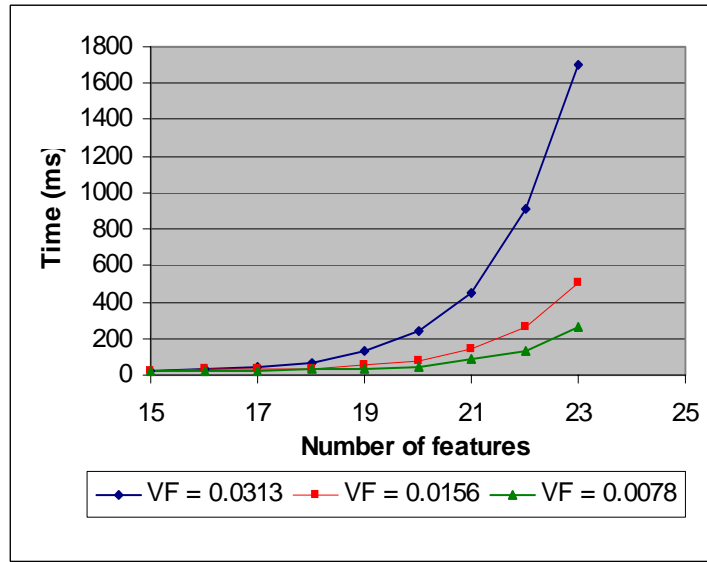

Figure 3. Empirical performance test for $products(M)$

7   Conclusion and Further Work

In this paper we set the basis for reasoning on SPL with features and attribute relations at the same time and in the same model using constraint programming.

There are some challenges we have to face in the near feature, namely: $i)$ extending our model to support dependencies such as a feature that *requires* or *excludes* another feature (e.g. video on demand requires $ADSL128$) that are also proposed in other

feature models ii) extending our current feature markup language to include extra–functional features iii) developing a case tool to validate our model on an industrial context, iv) performing a more rigorous validation of our implementation, studying the influences as well as the number of solutions, the types of relations, the number of features, and so on, v) comparing our work with others in the product configuration field[19].

References

1. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Coping with automatic reasoning on software product ines. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management*, November 2004.
2. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002.
3. J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 1th edition, 2000.
4. J. Bosch and H. Obbink. Proceedings of the 2nd Groningen Workshop on Software Variability Management. Technical Report to be published, University of Groningen, November 2004.
5. G. Chastek, P. Donohoe, K.C. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, June 2001.
6. P.C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.
7. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison–Wesley, may 2000. ISBN 0–201–30977–7.
8. A. van Deursen and P. Klint. Domain–specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
9. M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of theFifthInternational Conference on Software Reuse*, pages 76–85, Canada, 1998.
10. S. Jarzabek, Wai Chun Ong, and Hongyu Zhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.
11. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
12. K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature–oriented reuse method with domain–specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
13. K.C. Kang, J. Lee, and P. Donohoe. Feature–Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, July/August 2002.
14. 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.
15. K. Marriot and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
16. Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.

17. M. Shaw. Prospects for an engineering discipline of software. *IEEE Softw.*, 7(6):15–24, 1990.
18. M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *Proceedings of the Third Software Product Line Conference (SPLC04)*, San Diego, CA, 2004.
19. T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12(4):357–72, 1998.
20. D. Streitferdt, M. Riebisch, and I. Philippow. Details of formalized relations in feature models using ocl. In *Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS 2003), Huntsville, USA. IEEE Computer Society*, pages 45–54, 2003.
21. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.
22. J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), IEEE Computer Society*, pages 45–54, 2001.
23. A. Wasowski. Automatic Generation of Program Families by Model Restrictions. In *Proceedings of the Third Software Product Line Conference (SPLC04)*, San Diego, CA, 2004.