Potential of Integer Programming for Optimization Analysis of Extended Feature Models

Richard Heijblom
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands

a.r.heijblom@student.utwente.nl

ABSTRACT

In Software Product Line Engineering feature models are frequently used. These models are used to manage variability and commonality. In the past decades automatic analysis methods have been developed to facilitate information extraction of feature models. A popular technique is constraint programming. However this technique may fail in analyzing large models; in particular, optimization analysis will not be efficient, both in speed and memory usage. This problem is even worse for extended feature models, in which features could have attributes. Recently it was shown that optimization analysis could be performed with integer programming, which potentially can overcome the mentioned difficulty, but it is not used in any feature model analyzer. The goal of this paper is to address the potential of integer programming with respect to optimization analysis of extended feature models. This was investigated by comparing its performance with the performance of constraint programming on several extended feature models, with respect to speed.

Keywords

Optimization analysis, speed, extended feature model, constraint programming, integer programming.

1. INTRODUCTION

Software Product Line Engineering is all about generating a family of software products efficiently. In order to survey the family of products, commonalities and variabilities of products need to be determined. Feature models are a popular tool to manage these commonalities and variabilities [3,7,10].

A feature model describes a family of products in terms of features. A feature is a characteristic component of a product. A feature model itself is a tree composed of features and relations between those features, which describes all possible products at a certain stage in a software product line. In practice features itself could contain some information. Therefore, extended feature models were developed, where features could have attributes [3]. Because of the size, manual analysis of feature models is time consuming. Moreover, this method may not cover all possibilities or introduce wrong possibilities when errors are made. Therefore, there was a need for automatic analysis of feature models. Since then a lot of analysis methods have been developed [3]. This paper focuses on the optimization analysis, namely *'What is the optimal product of a given feature model?'*.

A popular technique for automatic analysis is constraint programming [1,2,3]. The feature model is transformed into a constraint satisfaction problem (CSP). Then off the shelf tools could analyze the CSP. It is possible to find optimal products using constraint programming, but in the case of large feature models this technique is not sufficient. Research suggests that large feature models (i.e. hundreds or thousands of features) could not be analyzed [1,2,4]. In the case of extended feature models, the attributes add an extra level of complexity [11]. Therefore, constraint programming is not powerful enough to analyze all of the extended feature models used in practice.

Another technique for automatic analysis is integer programming [5]. Integer programming is an optimization technique and used to solve so called integer problems (IP's). By definition it is not suitable to answer questions like *'What are the products of this feature model?'*, because there is nothing to optimize. Despite this drawback this technique may be very appropriate to find optimal products [6]. It has never been investigated whether or not integer programming is preferable in the case of optimization analysis. Therefore this paper aims at determining the possibilities and drawbacks of the two different techniques mentioned above with respect to optimization analysis. This was achieved by comparing the performance of a CSP solver and a IP solver on finding an optimal product of randomly generated extended feature models. This determined the potential of integer programming with respect to optimization analysis of extended feature models.

First a brief overview of feature models and optimization analysis is given in chapter 2. Then in chapter 3 related work is discussed. These two chapters lead to a definition of the purpose of this study described in chapter 4, and a research method described in chapter 5. In chapter 6 and 7 the results of this study are discussed.

2. BACKGROUND

2.1 Basic Feature Models

The first feature models were proposed by Kang et al. [10]. Although there are different notations of feature models, there is a common notion [3]. This notion states a feature model consists of hierarchically ordered features and relations among them, which describes a family of products, where a product is a non empty set of features. A formal definition is given below. To illustrate this notion, we use the example feature model of Benavides et al. [3]. Figure 1 contains an illustration of their feature model of a mobile phone.

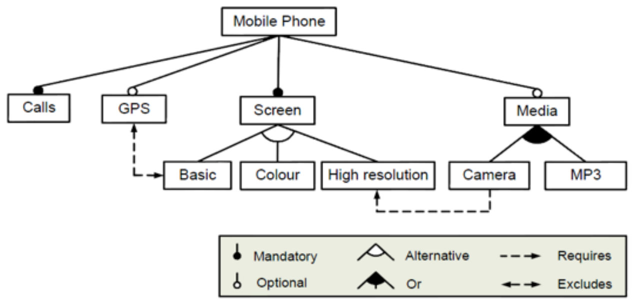

Figure 1. A feature model of a mobile phone

Definition feature model: A feature model is a four-tuple (F,T,P,C) where:

- F is a nonempty set of features.
- T is a tree structure of F. Each feature A in F except one feature called the root feature, has a unique parent feature $B \neq A$ in F. A is called a child feature of B.
- P is a function defined on F. For every feature A in F this function assigns a set of all possible combinations of child features of A. For every leaf feature this function assigns $\{\emptyset\}$.
- C is a set of crosstree relations.

As result of this definition one can conclude the following about the feature model in Figure 1: The set of features is {*Mobile Phone, Calls, GPS, Screen, Basic, Colour, High resolution, Media, Camera, MP3*}. The root feature is *Mobile Phone*, *Screen* is the parent of *Colour* and *Camera* is a child of *Media*. The children of *Mobile Phone* are *Calls*, *GPS*, *Screen* and *Media*. The features *Mobile Phone*, *Screen* and *Media* are the different parent features.

In a feature model (F,T,P,C), all the possible assignments of a function P are usually restricted to certain common assignments. These assignments are composed of four parent-child relations are listed below:

- Mandatory. In this relation if the parent feature is present in the product, the child feature is present too. In the given example, each product has the feature *Calls*.
- Optional. In this relation if the parent feature is present in the product, the child feature could be in the product. In the given example, a product could have the feature *GPS*. However, not every product has the feature *GPS*.
- Alternative. In this relation if the parent feature is present in the product, only one of the child features is in the product. In the given example, every product has a *High resolution* or a *Basic* or a *Color Screen*, but not two of them.
- Or. In this relation if the parent feature is present in the product, at least one of the child features is in the product. In the given example, every product with *Media* has a *Camera* or *MP3* or both of them.

The feature model in Figure 1 contains all four of these parent-child relations. For example *P*(*Media*) = {{*MP3*}, {*Camera*}, {*MP3*,*Camera*}} which is a or-relation. The graphical notation of these relations will be used in the rest of the paper.

Informally, the function P of a feature model (F,T,P,C) restricts the space of all possible products. Other restrictions on the space of products are called crosstree relations. It could be any relation between two or more features that is not a parent-child relation:

Definition crosstree relation: Given a feature model (F,R,P,C). A crosstree relation is a logical formula defined on a subset of F. This subset contains at least two elements.

A example of a crosstree relation in Figure 1 is *Camera requires High resolution*. The following two relations are typically used:

- Requires. If a feature A require feature B, then every product with feature A also has feature B. In the given example, every product with a *Camera* has a *High resolution Screen*. The logical formula for this relation is $A \Rightarrow B$.
- Excludes. If two feature excludes each other, then both features could not be in the same product. In the given example, no product exists with a *GPS* and a *Basic Screen*. The logical formula for this relation is $\neg(A \wedge B)$.

As stated above a feature model describes a family of products. Each product obeys the relations and restrictions described in that model. This leads to the definition of product:

Definition product: Given a feature model (F,T,P,C). A product S is a non empty subset of F in which:

- For each element A of S, if A is not the root feature, then the parent feature of A is an element of S too.
- If Q is an element of S, then the intersection of the set of all child features of Q and S is an element of $P(Q)$.
- Each statement in C is true, where each feature A in each statement is true if and only if A is element of S.

To illustrate this definition one can state the following about the feature model in Figure 1: {*Mobile Phone, Calls, Screen, Basic*} is a product. Another product is {*Mobile Phone, Calls, Screen, High resolution, Media, Camera*}. {*Mobile Phone, Calls, Screen, Basic, MP3*} and {*Mobile Phone, Screen, Basic*} are not products, because in the first set the child-feature *MP3* has no parent feature and in the second set the mandatory feature *Calls* is missing. The set {*Mobile Phone, Calls, Screen, Basic, Colour*} is not a product either, because in a product *Screen* can only have one child, but the set contains both *Basic* and *Colour*. For the reader who is interested in all the different products of the given model, should consult [3].

Definition void: A feature model is void if and only if the feature model does not represent any products.

In this paper we will use the term variability to indicate the number of different products of a given feature model. The higher the variability, the more different products a feature model represents. A void feature model is considered of the lowest variability.

2.2 Extended Feature Models

In some cases the basic feature model is too abstract. It could be necessary to include information about the features in the model. This piece of information is called an attribute of a feature. A feature model with attributes is called an extended feature model. Benavides et al. [3] concluded in their literature review that there is no consensus of how to use feature attributes. We adopt the idea of attributes of [11]: An attribute has a name and a domain, where the domain consists of the possible values the attribute could take. Thus an extended feature model is:

Definition extended feature model: An extended feature model is a five-tuple (F, T, A, P, C) where:

- (F, T, P, C) is a feature model.
- A is the set of attributes. An attribute is a three tuple (Q, N, D) where Q is an element of F (the feature which the attribute belongs to), N is the name and D is the domain of the attribute.

A product of an extended feature model almost equals the definition of a product of a basic feature model. The difference is that each feature in a product of an extended feature model has an instantiation for each attribute of that feature. That means every attribute in a product takes its value on its corresponding domain.

Definition product: Given an extended feature model (F, T, A, P, C). A tuple (S, I) where S is a nonempty subset of F and I is a set of attributes, is product if:

- S is a product of the feature model (F, T, P, C).
- For each element (Q, N, D) in A, if Q is an element of S then there is a unique (Q, N, E) in I, where E is a subdomain of D with a single value.
- For each element (Q, N, E) in I, Q is an element of S and (Q, N, D) is an element of A for some D which includes E.

This definition suggests that the space of products of an extended feature model is larger than the space of products of the corresponding basic feature model. This is generally true, because a product of a basic feature model can have one or more instantiations of attributes in an extended feature model.

In this paper, we classify the domains in three categories:

- Real interval. In this case the value of an attribute is an element of a certain bounded subinterval of the real numbers. For example, a possible attribute for *Calls* is *Frequency* with a value on the real domain $[0..100]$.
- Discrete interval. In this case the value of an attribute is an element of a certain bounded subinterval of the natural numbers. We use the notation $(a..b)$ for a interval with the minimum value a and the maximum value b. So $(3..7) = \{3,4,5,6,7\}$. For example, a possible attribute for *GPS* is *Version* with a value on the integer domain $(1..4)$.
- Enumeration. In this case the value of an attribute is a element of an enumeration. For example, a possible attribute for *Camera* is *Megapixels* with a value on the set $\{1,2,4,8,16\}$.

2.3 Optimization Analysis

One of the key questions that can be stated about feature models is: *'What is the optimal product of a given feature model?'*. The meaning of the word 'optimal' depends on the user of the product. Generally speaking, a product is better when that product is valued more. Then the optimal product is the product which the user assigns the highest value. This idea leads to the following definitions:

Definition Objective function: Let M be a feature model and let $P(M)$ be the set of all products described by M. Then a function $P(M) \rightarrow \mathbb{R}$ is an objective function.

In other words, an objective function is a function which assigns a real value to a every product of a certain feature model.

Definition Optimal product: Let M be a feature model and let $P(M)$ be the set of all products described by M. Let φ be an objective function defined on M. Then O is an optimal product with respect to $\varphi \Leftrightarrow \varphi(O) \geq \varphi(p)\ \forall p \in P(M)$.

Thus an optimal product maximizes a given objective function of a certain feature model. For example consider the feature model in Figure 1. If the objective function equals the number of features of the given product, then the optimal product is {*Mobile Phone, Calls, GPS, Screen, High resolution, Media, Camera, MP3*}.

Currently, optimization analysis could be performed with constraint programming or integer programming.

2.3.1 *Constraint programming*

Constraint programming (CP) is a set of techniques to solve constraint satisfaction problems (CSP's). A CSP consists of a set of variables and a set of constraints [1]. Each variable is defined on a certain domain. A constraint is a Boolean expression defined on some variables. If specific values for the variables evaluate a constraint true, then the constraint is satisfied, else the constraint is violated. The goal of solving a CSP is to find one or more solutions in which each variable has a value, such that this value is in the corresponding domain and all constraints are satisfied.

2.3.2 *Integer programming*

Integer programming [6] is a set of techniques to solve integer problems (IPs). An IP is a linear problem, except all variables are subjected to be integers. A linear problem is an optimization problem and consists of a set of numeric variables, a set of constraints and a linear goal function. These constraints are in the form of $\sum_i w_i \cdot x_i \gtreqless v$, where w_i en v are values and x_i are variables. $\gtreqless$ stands for $<, >, \leq, \geq$ or $=$. The goal of linear programming is to find a value for each variable, such that the goal function is maximized and all constraints are satisfied.

3. RELATED WORK

Benavides et al. [2] were the first who used constraint programming to reason on feature models. They provided a set of mapping rules to transform a feature model into a CSP. Although they did not provide mapping rules for attributes, they were able to transform an extended feature model into a CSP. Karates et al. [11] did provide these mapping rules. Moreover, they were able to map every relation into a part of a CSP. Both limited their feature models to have attributes with finite domains. As result a large number of feature models can be analyzed. Because of the mature field of constraint programming a wide variety of tools is available [11].

There were different performance analysis of the use of constraint programming to reason on feature models. The first analysis suggested an exponential growth in time when the features grow linearly [1,2]. Another analysis concluded roughly the same about memory usage and also stated that bigger feature models are a problem [4]. They also stated that constraint programming did not solve most problems in a reasonable amount of time. These analyses were based on determining all different products of a feature model.

Performance of optimization using constraint programming is not validated. The definition of the optimization operator [1] suggests that all products need to be checked to determine the optimal product. As stated above this suggests that optimization can be inefficient for large models, because a CSP solver has to cycle through all products.

Osman et al [12,13] also proposed a technique for optimization. However, their technique was developed for validating feature models. As side effect it is possible to do some optimization. This technique suffers the same problem as constraint programming: all product have to be checked in order to determine the optimal product. This problem was also pointed out by White et al. [16]. They addressed the optimization problem with an approximation [15,16]. Although they were able to analyze large feature models, the optimal products itself were usually not found.

Optimization analysis could also be done with integer programming. When given an linear problem, an optimal solution could be determined by the simplex algorithm. Although this algorithm performs very bad on certain classes of problems, this algorithm is generally fast. Van den Broek [5] showed that it is possible to transform a basic feature model into a IP. Therefore, the optimization can be done relatively easy [6]. However, the objective function and all relations in an extended feature model need to be linear in order to be transformable into a integer problem. This limits the set of feature models to which this technique can be applied on.

4. RESEARCH GOAL

The potential of integer programming is unknown. Also the performance of optimization using constraint programming is not clear.

Therefore, this paper aims at answering the following main question: What is the performance of integer programming and constraint programming with respect to finding an optimal product of extended feature models?

In order to answer that question, the main question is divided in five subquestions for each technique:

1. Which extended feature models can be solved in a reasonable amount of time?
2. How fast are extended feature models solved?
3. How is the solving time affected by the size of an extended model ?
4. How is the solving time affected by the composition of an extended model ?
5. How is the solving time affected by the type of the objective function? In other words, if another objective functions is chosen, how does it impact the solving time?

The meaning of 'solving' should be clear: finding an optimal product of a given extended feature model and a linear objective function defined on that model. This notion will be used in the rest of the paper.

Because of the currently known mapping rules, only extended feature models with mandatory-, optional-, or-, alternative-, requires- and excludes-relations were analyzed. Moreover, all attributes had a finite integer domain. In the case of extended feature models with real attributes, CSP solvers could not find an optimal product. Because of the infinite number of values an attribute could take, it is not possible to cycle through all products to find the optimal product. This suggests that linear programming is preferred. Therefore, extended feature models with real attributes were not investigated.

Extended feature models with enumeration attributes could be replaced by equivalent extended feature models without enumeration attributes. This is achieved by replacing each enumeration attribute by new child features, where each feature contains one single valued attribute of the enumeration. The relation between the new features equals the alternative-relation. Figure 2 illustrates the replacement of enumeration attributes. So every extended feature model with enumeration attributes could be replaced by an equivalent extended feature model without enumeration attributes. Therefore, this study was restricted to extended feature models where all attributes have a finite integer domain.

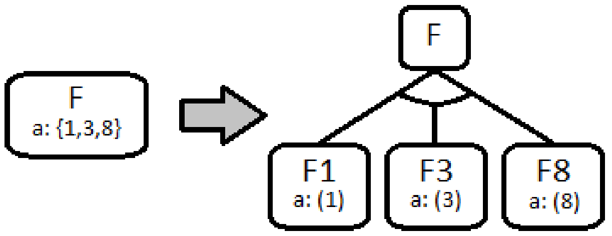

Figure 2: Replacement of an enumeration attribute

5. RESEARCH METHOD

To answer the five sub questions stated in the last chapter, a set of test feature models was generated. Each feature model in this set was analyzed by a CSP solver and an IP solver. Each solver had to find one optimal product of a given extended feature model. The time needed for the solvers were compared in order to classify groups of extended feature models and to analyze the two different techniques.

5.1 Components

An overview of the used components can be found in Figure 3.

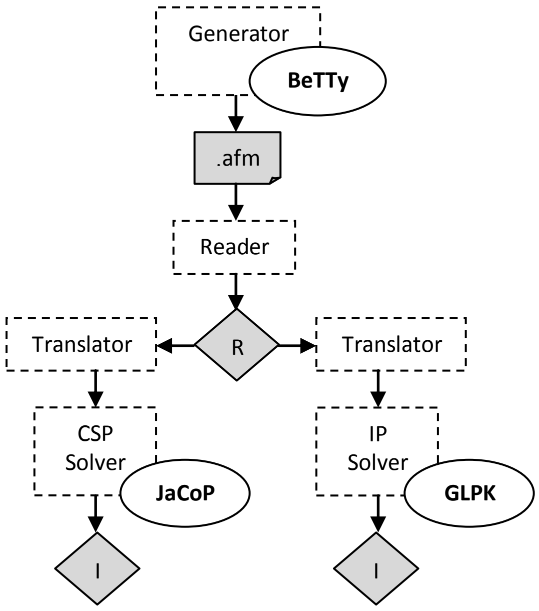

Figure 3: Overview components

Firstly, a generator was developed to generate the test feature models. The generation of the models itself was performed by BeTTy [14]. The generator managed the different feature models and saved the models in .afm files. The generated .afm files needed to be analyzed by the CSP solver and the IP solver. Because of the amount of feature models which have to be translated, a translator for both solvers were developed in Java. To avoid writing duplicate file readers, a single file reader was developed. This reader read the .afm files and translated it into another representation, denoted by R in Figure 3. The representation is basically a list of the parent-child and crosstree relations and a list of values of the different attributes. The developed translators could easily iterate through the lists in order to translate the extended feature model into a CSP or a IP. Both developed translators used the translation rules mentioned in [6]. Because the lack of mapping rules for objective functions and attributes, the objective function and the attributes were handled different in both translators.

The translator for constraint programming translated the objective function by adding a new variable called obj. A constraint $obj = \sum w \cdot a$ was also added, which restricted the variable obj to equal the used objective function (see next section). The translator for integer programming simply set the objective function as goal function.

The translator for constraint programming introduced a new variable for each attribute, restricted on the given domain. In order to prevent the attributes of features which are not part of the product to alter the value of the objective function, a constraint $IF\ f = 0\ THEN\ a = 0$ was added for each feature f and its attribute a. The translator for integer programming addressed both issues by adding a constraint $m \cdot f \leq a \ \leq n \cdot f$ for each feature f and its attribute a on the domain $(m..n)$.

Each solver took as input the representation of a extended feature model as discussed. The CSP Solver used JaCoP [9] for solving the CSP's and the IP Solver used GLPK [8] for solving the IP's. JaCoP and GLPK are both Java libraries. The solvers registered the solution and the time needed for solving, denoted by I in Figure 3. The first was used for validation of the developed programs and the latter was used for answering the main research question.

5.2 Test Models

The test models were the extended feature models generated by BeTTy. BeTTy automatically gave around 80% of all features in a model an attribute. Due to restrictions of the random feature model generator of BeTTy, each attribute ranged on the same domain. This domain was set to $(0..100)$. The generator also assigned a random number in this interval to each attribute. In order to assign different values to features, these randomly generated values were used as weight for an attribute in the objective function. How they were used depended on the test case. The objective function was a weighted sum of the attributes.

The test models will differ in three dimensions:

- Size. The size of a feature model equals the number of features the model contains.
- Composition. This means how the features in a feature model are related to each other in terms of parent-child relations.
- Constraint percentage. This percentage indicates the number of crosstree constraints compared to the size of a feature model. So a feature model with 80 features and 32 crosstree constraints has a constraint percentage of 40.

In order to generate test models with a certain composition, BeTTy requires percentages of all relations as input. In order to manage the size of the set of test feature models, the constraint percentages were restricted to be either 0, 30 or 60. We restricted all compositions to be in one of the five following categories.

- Balanced. In a feature model with a balanced composition around 30% of the parent child relations are mandatory and around 30% of the parent child relations are optional. The remaining 40% of the parent child relations are either alternative or or.
- Low variability. In a feature model with this composition around 80% of the parent child relations are mandatory and around 10% of the parent child relations are optional. The remaining 10% of the parent child relations are either alternative or or.
- High variability. In a feature model with this composition around 10% of the parent child relations are mandatory and around 80% of the parent child relations are optional. The remaining 10% of the parent child relations are either alternative or or.
- Low set participation. In a feature model with this composition around 45% of the parent child relations are mandatory and around 45% of the parent child relations are optional. The remaining 10% of the parent child relations are either alternative or or.
- High set participation. In a feature model with this composition around 10% of the parent child relations are mandatory or optional. The major 90% of the parent child relations are either alternative or or.

These restrictions brought the number of test feature models to fifteen different types for a fixed size. For each type ten random feature models were tested in order to achieve a better result. Therefore, for a fixed size, 150 different models were checked.

5.3 Execution and Measurement

All the models were solved on a Window 7 computer with an i7-3610QM Intel core at 2.3 GHz and 8 GB DDR3 RAM. The Java programs itself were executed in Eclipse Juno limited with a heap size of 2 GB.

Each solver automatically analyzed the test models. If a solver fails to find an optimal product in one minute, a 'fail' will be noted. Else the solving time was noted by using System.nanoTime(). This time limit was merely practical. Moreover, the solver needed to solve the models in a reasonable amount of time. Besides the solving time, the voidness of a test model was also noted.

5.4 Test Cases

The generator generated 1500 test models with a size of 100 until 1000, with steps of 100.

5.4.1 Case 0: Constraint programming strategies

JaCoP imposes the user to specify a search strategy. The Java library for JaCoP already contains several strategies. In this test case three strategies were chosen and their performance compared in order to find the best suitable strategy for the next two test cases. This performance was determined by the solving time of all the generated test models with a size lower than 600.

5.4.2 Case 1: All positive objective function

The objective function in this test case only consisted of positive terms. The generated attribute values were directly as weight in the objective function, because all of these values were positive. Both solvers analyzed the 1500 generated models separately. Because these solvers could easily solve the test models, another 1200 test models were generated and analyzed. These models ranged from 1500 until 5000 in size, with steps of 500. By performing this case the first four subquestions could be answered.

5.4.3 Case 2: Alternating objective function

In order to answer the last subquestion, another objective function was constructed. The values found in the .afm files were mapped into another weight. The mapping was as follows: if the weight is smaller than 51 the weight is unchanged, else the new weight equals 100 minus the old weight. This mapping introduces negative weights, and thus another objective function. Then the same test models were analyzed by the different solvers.

6. RESULTS

In the rest of the paper we use type numbers to indicate groups of test models with the same composition as follows:

0. Balanced and constraint percentage of 0.
1. Low variability and constraint percentage of 0.
2. High variability and constraint percentage of 0.
3. Low set participation and constraint percentage of 0.
4. High set participation and constraint percentage of 0.
5. Balanced and constraint percentage of 30.
6. Low variability and constraint percentage of 30.
7. High variability and constraint percentage of 30.
8. Low set participation and constraint percentage of 30.
9. High set participation and constraint percentage of 30.
10. Balanced and constraint percentage of 60.
11. Low variability and constraint percentage of 60.
12. High variability and constraint percentage of 60.
13. Low set participation and constraint percentage of 60.
14. High set participation and constraint percentage of 60.

All the solving times are average times shown in milliseconds, with a margin of a half millisecond. Only the most important results are listed below. The other results are listed in Appendix A.

6.1 Case 0

Figure 4 and 5 shows the solving times of the test models of type 0 using the CSP solver and three different strategies named IndomainMax (Max), IndomainMin (Min) and IndomainMiddle (Mid) with respect to the size of the models.

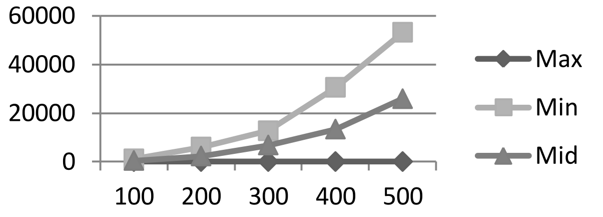

Figure 4: Solving times plotted against size (Case 1)

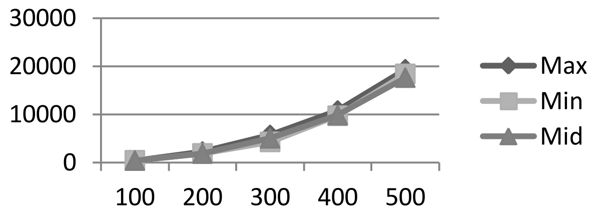

Figure 5: Solving times plotted against size (Case 2)

6.2 Case 1

Figure 6 shows the solving times of the test models of type 0 using the different solvers with respect to the size of the models.

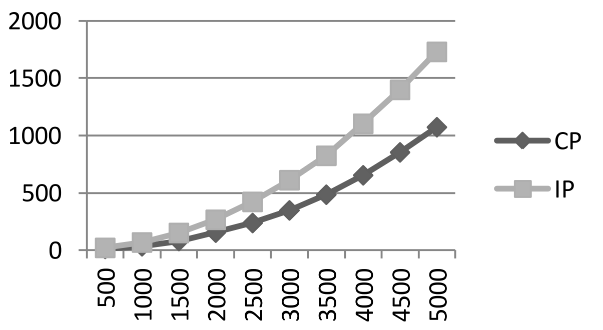

Figure 6: Solving times of both techniques

Figure 7 and 8 shows the solving times of the test models of the first five types using the different solvers with respect to the size of the models. Each curve in the graph is a type.

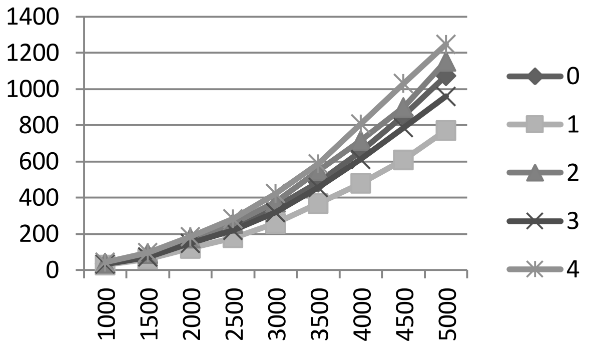

Figure 7: Solving times using constraint programming

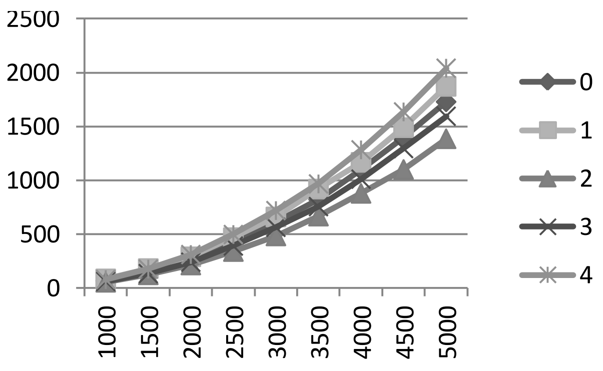

Figure 8: Solving times using integer programming

6.3 Case 2

Figure 9 shows again the solving times of the test models of type 0 using the different solvers with respect to the size of the models. This time the alternative objective function was used.

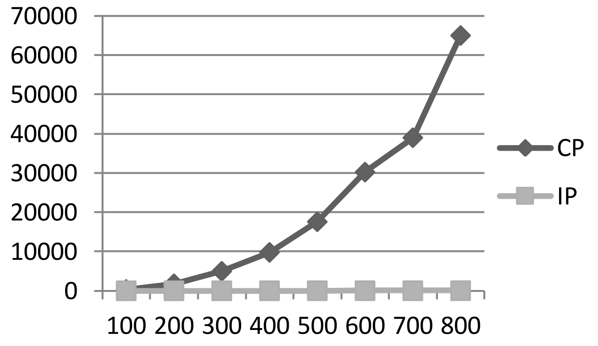

Figure 9: Solving times of both techniques

Table 1 shows the number of test models which could not be solved for each solver and test case. CP 1 stands for constraint programming case 1. Likewise IP 2 stands for integer programming case 2. The first column lists the different solvers and cases. The second column denotes the number of test models which could not be solved. The third column lists how many test models were analyzed. The last column denotes which percentage of all test models could not be analyzed

Table 1: Numbers and percentages of failures

setting	number	total	percentage
CP 1	138	2700	5,11
IP 1	0	2700	0
CP 2	80	1200	6,67
IP 2	0	1200	0

7. DISCUSSION

7.1 Case 0

The IndomainMax strategy starts with picking the highest value in a domain. After that the second highest value is chosen. The IndomainMin strategy is the counterpart of IndomainMax, which chooses the lowest value in a domain. The IndomainMiddle strategy combines the other two by first picking the middle value, then the highest value, then the lowest value in a domain. Figure 4 shows the speed compared to the size of the test models. Clearly, IndomainMax is the best strategy in for test case 1, which is not unlikely. In order to achieve a high objective function, as many features and as high as possible attributes values should be picked. This is exactly how IndomainMax works. The IndomainMiddle strategy firstly considered the middle values in domains. As result it took this strategy a factor 100 - 2000 more time to solve the test models. The IndomainMin strategy is even worse. Probably this strategy considered almost all lower valued products before trying to add a feature. Moreover, Figure 4 suggests that the IndomainMiddle and the IndomainMin strategy's solving time grows exponentially when the size grows linearly. Without considering test models of type 7, The IndomainMin strategy already fails to solve some test models with a size of 500. Therefore, for test case 1 the IndomainMax strategy was chosen.

By setting the objective function to the one mentioned in section 5.4.3, all three strategies took around the same solving time according to Figure 5. The IndomainMiddle strategy was overall slightly faster, so this strategy was chosen for test case 2.

7.2 Case 1

7.2.1 Analysis of solvability

By analyzing the results in Table 1, we conclude that there is a difference between constraint programming and integer programming. With integer programming all test models were solved in the one minute time limit. With constraint programming however 138 models could not be solved in the time limit. 127 of those model were of type 7 and the remaining 11 of those models were of type 12. Although this failure is only around 5%, some failures were early encountered. 5 test models with a size of 200 already caused failures for the CSP solver, while the IP solver did not experienced any problems. By judging the solvability, integer programming seems to be preferred over constraint programming

7.2.2 Analysis of speed

Figure 6, 7 and 8 show the average solving time per test model composition. By basically comparing each average time, constraint programming is superior compared to integer programming. Considering only the test models with a constraint percentage of 0, the CSP solver is about a factor 2 faster than the IP solver. Only 7% of the test model could be faster solved with integer programming. All of these models are of type 7 or type 12. Most of those models are the same which were considered to failure to solve for the CSP solver. On the other hand, all test models could be solved in less than 4.500 seconds by using integer programming. Both solvers suggest to grow exponentially when the size grows linear. In most cases, constraint programming seems to be preferred over integer programming.

7.2.3 Influence of voidness

While generating extended feature models, BeTTy does not make any guarantees about feature models which represents some products. Therefore, BeTTy can generate void feature models. Considering the test models of type 5 until 14, al lot of these models are actually void. Moreover, all the generated test models of type 6 and 11 are void. This could be clarified by the composition. While trying to make a product, if someone encounters two mandatory features which exclude each other, then the model is void if those two feature should be in the product. Models of type 6 and 11 contain a large number of mandatory relations and exclude relations. Most test models with a constraint percentage of 60, are void probably due the large number of exclude relations. The test models of type 12 did not suffer voidness. Probably due the large number of optional features it is possible to circumvent the exclude relations and makes it easier to create products.

By comparing the solving times, Appendix A clearly shows a speed gain for void models by using constraint programming. The voidness of most test models was discovered in less than a half millisecond. It took the CSP solver at most 2 milliseconds to discover voidness. The results show that the IP solver also solved void test models faster than other test models. This gain is not as extreme as the gain by constraint programming, but it suggests that the IP solver is a factor two faster when analyzing a void test model. By judging this aspect, constraint programming seems to be preferred over integer programming by analyzing void feature models.

7.2.4 Influence of constraint percentage

By considering the influence of the constraint percentage, the results show that constraint programming benefits from an increasing constraint percentage. The comparisons of the nonvoid test models 8, 9, 10, 12, 13 and 14 with the respective test models 3,4 and 5 suggest that extended feature models with a high constraint percentage are faster solvable than other with a low constraint percentage. Crosstree relations restrict the space of possible products and therefore they help to speed up the CSP solver. Test models of type 12 and 7 confirm this notion, but test models of type 2 suggest otherwise. Probably the lack of any crosstree constraint allows the CSP solver to find the optimal product quickly, because a selection of a feature never has to be revoked. Generally, an increase in constraint percentage seems to reduce the solving time while using constraint programming.

The same test models mentioned in the last paragraph give a consistent view for integer programming. The IP solver suffers from an increase in constraint percentage in terms of solving speed. The solving time may be a factor 3 slower due the crosstree constraints. Each crosstree relation introduces more space in the matrix solved by the simplex algorithm. This fact may clarify the penalty in the speed. So an increase in constraint percentage seems to increase the solving time while using integer programming.

7.2.5 Influence of variability

The test models of type 0, 1 and 2 and Figure 7 and 8 confirm the notions of 7.2.4. An increase in mandatory relations seems to benefit the CSP solver, but an increase in optional relations seems to slow the solving time. Feature models with many mandatory relations have generally a low variability and therefore a small possible product space. On the other hand, feature models with many optional relations increases the variability and therefore a larger possible product space. The larger this space, the longer the CSP solver is takes to find an optimal product. The increase in variability seems to benefit the IP solver. The test models with a lot of optional relations were fasted solved than the balanced test models, which were faster solved than the test models with a lot of mandatory relations.

7.2.6 Influence of set participation

Considering Figure 7 and 8, the occurrence of alternative and or relations influences both solvers in the same way. Considering models of type 0, 3 and 4, both solvers solved the test model with a few alternative and or relations faster than the balanced test models. The speed gain is about 8%. The solving time of the test models with a lot of alternative and or relations was about 20% longer than the ones of the balanced test models. The IP solver seems to be slightly more stable than the CSP solver.

7.3 Case 2

Figure 9 contains the solving times of test case 2. By alternating the objective function, the IP solver seems to be quite stable. For the most test models, the solving time needed is even slightly lower than the test models with the objective function of case 1. This suggests that the IP solver can easily handle objective function with positive and negative terms.

The CSP solver cannot handle both positive and negative terms in the objective function. Because of this test models with a size larger than 800 were not even analyzed. The time needed to solve the test models with a constraint percentage of 0 resembles the solving times of the IndomainMiddle strategy of case 0. The IP solver is over a 1000 times faster than the CSP solver. Twice as many test models could not be solved by using constraint programming as result of the alternated objective function. These failures also occurred in the models of type 0, 1, 2, 3 and 4, while the CSP solver did not experience any trouble with these test models five times larger in test case 1. The CSP solver still recognized the void test models extremely fast, but also test models with a high constraint percentage took more time than using integer programming.

8. CONCLUSION AND FURTHER WORK

Constraint programming can be a very powerful technique when tackling optimization analysis, when the right information is known. A specific search strategy needs to be determined in order to use the speed of a CSP solver. Overall, integer programming is preferred, because an IP solver does not need this information. Besides a small penalty for speed, integer programming looks like a reliable technique when solving extended feature models.

The used CSP solver fails in some cases to solve an extended feature model in a reasonable amount of time. None of the test models was a problem for the IP solver.

The speed of CSP solver was overall higher than the speed of the IP solver. Only in a few specific cases the IP solver was faster. Both techniques offer the same time complexity regarding to the size of an extended feature model. The time needed grows exponentially when the size grows linearly.

The variability of an extended feature affects the solving time. The CSP solver seems to benefit from a low variability, while the IP solver seems to benefit from a high variability. The solving time of a void feature model is extremely low while using constraint programming.

The solving time while using integer programming is stable under changes of the objective function, while the CSP solver suffers extremely in speed while solving models with both positive and negative terms in the objective function.

However, the mapping from an extended feature model to an integer problem is not complete. The mapping mentioned in [5] and in this paper is sufficient to map the six most used relations and attributes, but the mapping of all relations mentioned in [11] are not yet defined. Furthermore, existing software could be upgraded to use integer programming for optimization analysis.

9. REFERENCES

[1] D. Benavides, A. Ruiz-Cortés and P. Trinidad. Automatic reasoning on feature models. In *17th Conference on Advanced Information Systems Engineering*, 2005.

[2] D. Benavides, A. Ruiz-Cortés and P. Trinidad. Using constraint programming to reason on feature models. In *17th International Conference on Software Engineering and Knowledge Engineering,* 2005.

[3] D. Benavides, S. Segura and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. In *Information Systems 35*, 2010.

[4] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[5] P. van den Broek. Optimization of product instantiation using integer programming. In *Proceedings of the 14th International Software Product Line Conference*, 2010.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. In *Introduction to algorithms*, 2009.

[7] K. Czarnecki and U. Eisenecker, Generative programming: methods tools and applications. Addison-Wesley, 2000.

[8] GLPK, http:// www.gnu.org/software/glpk/, accessed April 2013.

[9] JaCoP, http:// www.jacop.osolpro.com, accessed March 2013.

[10] 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*, 1990.

[11] A. Karatas, H.Oguztüzün and A. Dogru. Mapping extended feature models to constraint logic programming over finite domains. In *Proceedings of the 14th International Software Product Line Conference*, 2010.

[12] A. Osman, S. Phon-Amnuaisuk and C.K. Ho. Knowledge based method to validate feature models. In *1st International Workshop on Analyses of Software Product Lines*, 2008.

[13] A. Osman, S. Phon-Amnuaisuk, and C.K. Ho. Using first order logic to validate feature model. In *3rd International Workshop on Variability Modeling of Software-intensive Systems*, 2009.

[14] S. Segura, J.A. Galindo, D. Benavides, J.A. Perejo and A. Ruiz-Cortés. BeTTy: Benchmarking and Testing on the automated analysis of feature models. In *6th International Workshop on Variability Modeling on Software-intensive Systems,* 2012.

[15] J. White, B. Doughtery and D. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. In *Journal of Systems and Software*, 2009.

[16] J. White and D. Schmidt. Filtered cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. In *1st International Workshop on Analyses of Software Product Lines*, 2008.

APPENDIX

A. RESULTS

This section contains most of the results of this study. The information is ordered in tables. In each table the first row denotes the different sizes of test models. The first column denotes the different types of test models. The other cells contain a average runtime in milliseconds. Each grey cell denotes that all test models of that size and type were void. The absence of any number means that the solver could not solve any of the test models of that size and type.

A.1 Case 0

Table 2 contains the average runtimes of the three different strategies using constraint programming for determining the strategy for test case 1. Table 2 contains one extra dimension. The second row denotes which strategy was used. Max denotes IndomainMax, Min denotes IndomainMin and Mid denotes IndomainMiddle. The runtimes in this tables only includes the first case. The different runtimes for the second case resembles each other and are therefore not recorded in this paper. Only the runtimes with IndomainMiddle strategy is recorded in section A.3

A.2 Case 1

Table 3 and 4 contains the average runtimes of the solvers in test case 1.

A.3 Case 2

Table 5 and 6 contains the average runtimes of the solvers in test case 2.

Table 2: Average solving times using different strategies (Case 1)

	100			**200**			**300**			**400**			**500**		
	Max	**Min**	**Mid**	**Max**	**Min**	**Mid**	**Max**	**Min**	**Mid**	**Max**	**Min**	**Mid**	**Max**	**Min**	**Mid**
0	4	1112	459	4	5983	2301	5	12661	6813	7	30520	13354	12	53129	25839
1	2	1122	410	2	6975	2920	4	17904	7344	6	31870	13884	9	50835	23410
2	2	736	399	2	4160	2348	4	12067	6479	7	28600	14125	10	42510	24768
3	1	1075	430	2	6675	2571	3	14432	6947	5	23645	13196	8	42926	21372
4	1	1368	478	3	6541	2952	7	18083	9117	6	32176	16344	11	54229	27857
5	1	1	1	0	0	0	1	22	6	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	13	495	364	59999			1	160	71	0	0	0			
8	0	11	3	0	0	0	3	640	184	0	0	0	0	0	0
9	0	37	10	0	0	0	1	483	166	0	0	0	1	56	5
10	0	0	0	0	0	0	0	0	0	2	18	6	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	405	624	536	1	4	1	1467	4184	3449	1	159	64	45	724	360
13	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: Average solving times using constraint programming

	100	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1500**	**2000**	**2500**	**3000**	**3500**	**4000**	**4500**	**5000**
0	4	4	5	7	12	14	20	23	27	35	81	154	238	348	485	654	854	1073
1	2	2	4	6	9	10	12	16	21	27	61	119	178	257	367	479	608	771
2	2	2	4	7	10	12	17	22	29	37	91	176	257	374	548	716	898	1153
3	1	2	3	5	8	10	14	26	26	33	72	147	219	317	456	611	787	959
4	1	3	7	6	11	14	19	22	34	43	96	184	283	423	587	807	1031	1247
5	1	0	1	0	0	1	0	2	5	4	1	0	1	1	1	1	2	2
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
7	13	59999	1	0			5675					3						
8	0	0	3	0	0	2	0	1	0	0	3	1	1	1	1	1	1	10
9	0	0	1	0	1	1	0	0	1	3	1	1	1	6	1	1	1	2
10	0	0	0	2	0	0	0	1	2	0	2	6	1	1	1	1	1	1
11	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	2
12	405	1	1467	1	45	7	5	30080	50	1	18	45	15	17	6	57		90
13	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	8	1	2
14	0	0	0	0	0	0	0	0	2	1	0	4	1	1	1	1	2	1

Table 4: Average solving times using integer programming

	100	200	300	400	500	600	700	800	900	1000	1500	2000	2500	3000	3500	4000	4500	5000
0	3	6	10	15	21	32	36	46	56	68	151	266	422	611	824	1103	1399	1730
1	4	6	10	20	24	32	41	55	67	86	182	291	469	661	920	1168	1490	1873
2	3	4	7	14	15	21	30	36	44	55	124	215	340	485	669	880	1102	1387
3	3	5	8	17	21	24	33	41	51	63	135	241	391	565	757	1012	1296	1596
4	3	5	10	20	23	31	41	54	68	80	177	312	498	720	968	1284	1636	2042
5	4	2	21	9	13	61	20	108	127	169	100	166	236	366	492	681	820	1034
6	2	5	8	14	21	28	35	49	59	74	161	289	455	654	862	1110	1432	1757
7	2	4	17	4	25	34	79	163	65	210	281	1106	833	662	1980	4159	3524	3785
8	2	3	16	7	10	56	19	97	25	31	316	158	208	298	415	600	694	3696
9	3	5	26	8	50	67	32	28	98	186	106	178	339	1702	617	727	1043	1262
10	1	8	5	36	18	25	33	122	160	68	432	757	417	570	772	1010	1327	1619
11	2	5	9	16	23	33	43	56	74	91	192	351	549	783	1049	1393	1742	2192
12	4	12	70	36	47	60	84	214	156	178	355	843	1014	1531	2107	2617	3312	4098
13	4	12	6	10	13	21	29	36	50	48	128	219	360	529	678	2896	1157	1408
14	1	4	9	14	20	27	38	52	188	151	179	897	531	698	942	1266	1601	1903

Table 5: Average solving times using constraint programming

	100	200	300	400	500	600	700	800
0	341	1787	4971	9781	17578	30219	38983	
1	317	2110	5244	9649	15901	23998	35319	49160
2	339	1694	4657	10005	17011	25847	36763	52685
3	355	2022	4915	9292	15890	24033	34576	47800
4	390	2114	6743	11687	19713	32091	48274	53089
5	1	0	6	0	0	2	1	5
6	0	0	0	0	0	0	0	0
7	223		57	0			3701	
8	2	0	140	0	0	29	0	49
9	8	0	115	0	5	2	0	0
10	0	0	0	4	0	0	0	2
11	0	0	0	0	0	0	0	0
12	505	1	19316	57	242	723	1213	7005
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0

Table 6: Average solving times using integer programming

	100	200	300	400	500	600	700	800
0	3	6	8	12	16	22	26	35
1	2	6	9	14	18	27	34	45
2	1	3	5	7	10	15	20	24
3	2	4	6	9	14	18	24	30
4	2	4	8	12	17	24	31	41
5	2	2	14	9	11	44	20	76
6	2	4	8	13	20	28	35	49
7	2	3	11	4	16	21	52	121
8	2	2	11	7	10	40	19	64
9	2	3	18	8	35	48	32	28
10	1	6	5	27	17	25	33	86
11	2	5	9	15	22	33	43	56
12	3	8	44	23	29	39	55	194
13	3	8	6	10	13	21	29	36
14	1	3	8	12	19	27	38	51