# Elimination of Constraints from Feature Trees

Pim van den Broek
*Department of Computer Science,*
*University of Twente*
*P.O. Box 217,*
*7500 AE Enschede,*
*The Netherlands*
*pimvdb@ewi.utwente.nl*

Ismênia Galvão
*Department of Computer Science,*
*University of Twente*
*P.O. Box 217,*
*7500 AE Enschede,*
*The Netherlands*
*i.galvao@ewi.utwente.nl*

Joost Noppen
*Département Informatique,*
*École des Mines,*
*4, rue Alfred Kastler,*
*F - 44307 Nantes cedex 3,*
*Nantes,*
*France*
*johannes.noppen@emn.fr*

## Abstract

*We present an algorithm which eliminates constraints from a feature model whose feature diagram is a tree and whose constraints are "requires" or "excludes" constraints. The algorithm constructs a feature tree which has the same semantics as the original feature model. The computational complexity of the algorithm is exponential in the number of constraints, but linear in the number of features. The algorithm allows to efficiently compute properties of product lines whose feature model consists of a feature tree and a small number of "requires" and "excludes" constraints. An executable specification of the algorithm is given in the functional programming language Miranda.*

## 1. Introduction

Feature models are used to specify the variability of software product lines [1,2]. A feature model consists of a feature diagram and a (possibly empty) set of constraints. The feature diagram is either a tree or a rooted directed acyclic graph (RDAG). To compute properties of the specified software product line is easy in case the feature diagram is a tree and there are no constraints; one simply writes recursive functions on trees. In case the feature diagram is a RDAG or there are constraints, computing properties of the described software product line is much more difficult. In a number of approaches in the literature, feature models are mapped to other data structures: Benavides et al. [3] use Constraint Satisfaction Problems, Batory [4] uses Logic Truth Maintenance Systems and Czarnecki and Kim [5] use Binary Decision Diagrams.

In this paper we present an approach which uses feature trees as the basic data structure, thereby staying as close as possible to the problem statement. We consider the case where the feature diagram is a tree, and there are the usual "requires" and "excludes" constraints. We present an algorithm which eliminates the constraints, and delivers a feature tree which is equivalent to the former feature tree with constraints. Properties of the software product line can then be computed by recursive functions on feature trees.

In the next section we provide some preliminary definitions. In section 3 we provide some auxiliary algorithms. In section 4 we present the algorithm to eliminate constraints. In section 5 we discuss the computational complexity of our algorithm, and show that its complexity is exponential in the number of constraints, but linear in the number of features. In an appendix we give a complete executable specification of our algorithms in the functional programming language Miranda.

## 2. Preliminaries

The feature models in this paper consist of a feature tree and a set of constraints. A feature tree is a tree whose nodes are called features. There are three types of nodes: MandOpt features, Or features and Xor features.

A MandOpt feature has two lists of subfeatures, called mandatory and optional subfeatures respectively. Or features and Xor features have 2 or more subfeatures. A leaf of the tree is a MandOpt feature without subfeatures.

A constraint has either the form "F1 requires F2" or "F1 excludes F2"

The semantics of such a feature model is a set of products, where each product is a set of features which occur in the tree [6]. A product belongs to the semantics of the feature model if and only if it satisfies the constraints from the tree as well as the explicit constraints.

A product satisfies the constraints from the tree if:

- It contains the root of the tree.
- For each feature except the root in the product, the product also contains its parent feature.
- For each MandOpt feature in the product, the product also contains all its mandatory subfeatures.
- For each Or feature in the product, the product also contains one or more of its subfeatures.
- For each Xor feature in the product, the product also contains exactly one of its subfeatures.

A product satisfies a constraint "F1 requires F2" when, if it contains F1 it also contains F2. A product satisfies a constraint "F1 excludes F2" when it does not both contain F1 and F2

Just for the ease of writing concise algorithms, we assume the existence of a special feature tree NIL, which cannot occur as subtree of other trees, and which has no products.

## 3. Auxiliary algorithms

In this section we present two auxiliary algorithms, which deal with commitment to a feature and deletion of a feature, respectively.

The first auxiliary algorithm computes, given a feature tree T and a feature F, the feature tree T(+F), whose products are precisely those products of T which contain F. The algorithm transforms T into T(+F) by means of the following steps:

1. If T does not contain F, the result is NIL.
2. If F is the root of T, the result is T.
3. Let the parent feature of F be P.
   - If P is a MandOpt feature and F is an optional subfeature, make F a mandatory subfeature of P.
   - If P is an Xor feature, make P a MandOpt feature which has F as single mandatory subfeature and has no optional subfeatures. All other subfeatures of P are removed from the tree.
   - If P is an Or feature, make P a MandOpt feature which has F as single mandatory subfeature. and has all other subfeatures of P as optional subfeatures.
4. GOTO step 2 with P instead of F.

The second auxiliary algorithm computes, given a feature tree T and a feature F, the feature tree T(-F) whose products are precisely those products of T which do not contain F. The algorithm transforms T into T(-F) by means of the following steps:

1. If T does not contain F, the result is T.
2. If F is the root of T, the result is NIL.
3. Let the parent feature of F be P.
   - If P is a MandOpt feature and F is a mandatory subfeature of P, GOTO step 2 with P instead of F.
   - If P is a MandOpt feature and F is an optional subfeature of P, delete F.
   - If P is an Xor feature or an Or feature, delete F; if P has only one remaining subfeature, make P a MandOpt feature and its subfeature a mandatory subfeature.

## 4. Elimination of constraints

Let a feature model be given by a feature tree T and a constraint "A requires B". We want to construct a feature tree whose products are those products of T which contain B when they contain A. This set of products is the union of the product sets of T(+B) and T(-A-B). Here T(-A-B) is a shorthand for (T(-A))(-B). The product sets of T(+B) and T(-A-B) are disjoint. So the required feature tree can be obtained by taking a new Xor feature as root which has T(+B) and T(-A-B) as subfeatures. The algorithm to eliminate "A requires B" from T is:

Construct T(+B) and T(-A-B).
If both trees are not equal to NIL, then the result consists of a new root, which is an Xor feature, with subfeatures T(+B) and T(-A-B). If T(-A-B) is equal to NIL, then the result is T(+B). If T(+B) is equal to NIL, then the result is T(-A-B).

Now let a feature model be given by a feature tree T and a constraint "A excludes B". We want to construct a feature tree whose products are those products of T which do not contain both A and B. This set of products is the union of the product sets of T(-B) and T(-A+B). Moreover, the product sets of T(-B) and T(-A+B) are disjoint. So the required feature tree can be obtained by taking a new Xor feature as root which has T(-B) and T(-A+B) as subfeatures. The algorithm to eliminate "A excludes B" from T is:

Construct T(-B) and T(-A+B).
If both trees are not equal to NIL, then the result consists of a new root, which is an Xor feature with subtrees T(-B) and T(-A+B). If T(-B) is equal to NIL, then the result is T(-A+B). If T(-A+B) is equal to NIL, then the result is T(-B).

Note that the feature trees obtained by these algorithms have the property that features may occur more than once. However, multiple occurrences are in different subtrees of an Xor feature, so products do not have multiple occurrences of features.

When there is more than one constraint, these constraints may be eliminated in sequel. The auxiliary algorithms of section 3 then should be modified by adding a repetition over multiple occurrences of features in a tree.

The efficiency of the algorithms presented above may be improved in two ways:

- Instead of eliminating a constraint from the whole tree, eliminate the constraint from the smallest subtree which contains A and B.
- Perform dynamic programming: keep track of identical subtrees, and perform identical operations on identical subtrees only once, using memoization.

## 5. Example

In this section we provide a simple example to illustrate the result of the algorithms in the previous sections. Consider the feature tree T in figure 1.

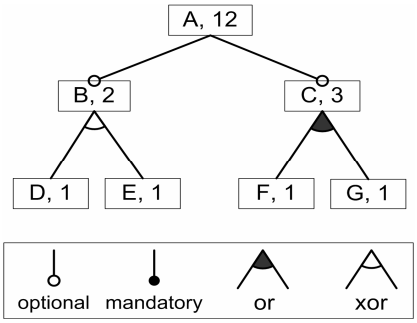


Figure 1. Example feature tree

Here the numbers indicate for each feature the number of products which correspond to its subtree. These numbers are calculated with the following straightforward recursive algorithm:

- For a MandOpt feature, the number of products is the product of the numbers of products for mandatory subfeatures, and the numbers of products incremented by 1 for optional subfeatures.
- For an Or feature, the number of products is 1 less then the product of the numbers of products of its subfeatures incremented by 1.
- For an Xor feature, the number of products is the sum of the numbers of products of its subfeatures.

Note that the number of products of an Or feature equals the number of features of a MandOpt feature with optional nodes only, minus one. A more formal, less verbose, definition of this algorithm is given in the appendix.

Now suppose there is an additional constraint: "D requires F". The algorithm of section 4 which eliminates this constraint from T gives the feature tree of figure 2:

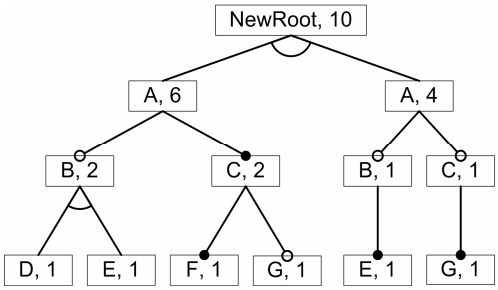


Figure 2. Example feature tree,
constraint "D requires F" eliminated

The root of this feature tree is a new Xor feature; its left subtree is T(+F) and its right subtree is T(-D-F). Again, the number of products, calculated with the algorithm above, are shown for each feature.

Now suppose there is an additional constraint : "D excludes F". The algorithm of section 4 which eliminates this constraint from T gives the feature tree of figure 3:

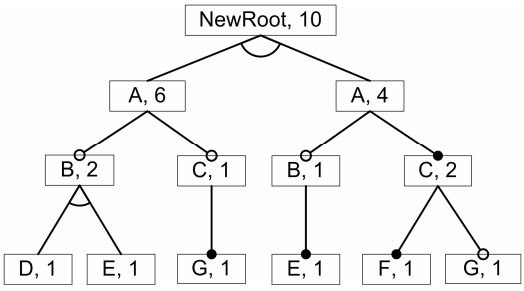


Figure 3. Example feature tree,
constraint "D excludes F" eliminated

The root of this feature tree is a new Xor feature; its left subtree is T(-F) and its right subtree is T(-D+F). Again, the number of products, calculated with the algorithm above, are shown for each feature.

## 6. Computational complexity

The algorithms given in section 4 clearly have linear time complexity. However, elimination of a constraint in the worst case doubles the size of the tree. Therefore, in the worst case, the size of the resulting tree will be exponential in the number of constraints. Algorithms which compute properties of a product line by first eliminating constraints from a feature tree will therefore always have exponential worst case time complexity. However, exponential computational complexity is inevitable, since, as we will show below, the decision problem whether or not a feature tree with constraints has at least one product, is NP-complete. So, there is no hope for an algorithm with polynomial computational complexity. Transformation of the feature model to a Binary Decision Diagram does not help, since this transformation itself has exponential computational complexity. An advantage of our approach is that, when the number of constraints is small, the algorithms will certainly be feasible. For instance, the algorithm which computes the number of products, given in the previous section, belongs to the complexity class O(N*2M), where N is the number of features in the feature tree and M is the number of constraints.

Now we will prove that the problem whether or not a feature model which is given by a feature tree and a set of constraints has at least one product is NP-complete. We do this by showing that the satisfiability problem SAT, which is NP-complete, can be reduced to our problem in polynomial time. This approach is similar to the approach by Schobbens et al. [6], who show that the corresponding problem with RDAGs instead of trees is NP-complete. SAT is the problem whether or not a Boolean expression which only contains Boolean variables and their negations, and which is in conjunctive normal form, can be satisfied by assigning Boolean values to the variables. An example expression is $(X \vee Y) \wedge (\neg X \vee \neg Y) \wedge (X \vee \neg Y)$. For this expression, we construct the feature tree in figure 4. For each clause in the expression the root feature has a mandatory subfeature. Each of these subfeatures is a Or feature, having each of its literals as subfeature.

The expression is satisfiable if and only if the feature tree has a product without contradictions. Here contradiction in a product means that for some variable

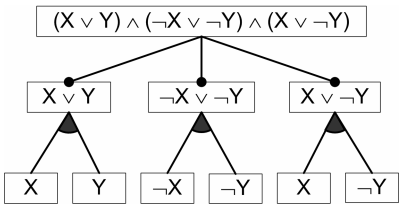


Figure 4

V, the product contains both V and ¬V. Products with contradictions can be excluded by introducing constraints. For each occurrence in the tree of features V and ¬V for some variable V we add the constraint that these features be mutually exclusive. In the example of figure 4, there are 4 such constraints. Since this construction of the feature model only requires polynomial time, this proves that our problem is NP-complete.

Here it is interesting to note that it is not the presence of the constraints which makes the problem NP-complete. If feature trees would only contain MandOpt features, the problem has polynomial computational complexity. This can be shown by reducing it to 2SAT, the satisfiability problem where each conjunction contains only 2 literals, which has polynomial computational complexity.

## 7. Conclusion

We have presented algorithms to eliminate "requires" and "excludes" constraints from a feature model whose feature diagram is a tree, by constructing a feature tree which has the same semantics as the original feature tree with constraints. These algorithms allow to efficiently compute properties of product lines whose feature model consists of a feature tree and a small number of "requires" and "excludes" constraints.

## 8. Acknowledgments

This work is supported by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

## 9. References

[1] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990).

[2] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods Tools and Applications*, Addison-Wesley (2000).

[3] D. Benavides, P. Trinidad and A. Ruiz-Cortés, "Automated Reasoning on Feature Models", in: O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, Lecture Notes in Computer Science 3520, Springer-Verlag Berlin Heidelberg, 2005, pp. 491-503.

[4] D. Batory, "Feature Models, Grammars, and Propositional Formulas", in: H. Obbink and K. Pohl (eds.): Software Product Lines Conference 2005, Lecture Notes in Computer Science 3714, Springer-Verlag Berlin Heidelberg, pp. 7-20, 2005.

[5] K. Czarnecki and P. Kim, Cardinality-based Feature Modeling and Constraints: A Progress Report, in: Proceedings of the International Workshop on Software Factories, OOPSLA 2005, 2005. http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

[6] P.-Y. Schobbens, P. Heymans, J.-Chr. Trigaux and Y. Bontemps, "Generic Semantics of Feature Diagrams", *Computer Networks* 51, 2007, pp. 456–479.

[7] D. Turner, Miranda: a non-strict functional language with polymorphic types, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science Vol 201, J.-P. Jouannaud (ed.), Springer-Verlag, Berlin, Heidelberg, 1985, pp. 1-16.

## Appendix

In this appendix we provide a complete executable specification of our algorithms in the functional programming language Miranda [7].

First we give some type definitions. Here `MandOpt nm ms os` stands for a MandOpt feature with name `nm`, `ms` is the list of its mandatory features and `or` is the list of its optional features.

```
name == [char]
tree ::= MandOpt name [tree] [tree] |
         Or name [tree] |
         Xor name [tree] |
         NIL
constraint ::= Requires name name |
               Excludes name name
feature_model == (tree,[constraint])
```

The expression `delete nm ft` is a feature tree whose products are the products of the feature tree `ft` which contain the feature with name `nm`.

```
delete :: name -> tree -> tree
delete nm NIL = NIL
delete nm (MandOpt n ms os)
  = NIL, if n=nm \/ member ms2 NIL
  = MandOpt n ms2 (filter (~= NIL) os2),
                                  otherwise
    where
    ms2 = [delete nm m|m<-ms]
    os2 = [delete nm o|o<-os]
delete nm (Xor n fts)
  = NIL, if n=nm
  = MandOpt n fts2 [], if #fts2 < 2
  = Xor n fts2, otherwise
    where
    fts2 = filter (~= NIL)
                  [delete nm ft|ft<-fts]
delete nm (Or n fts)
  = NIL, if n=nm
  = MandOpt n fts2 [], if #fts2 < 2
  = Or n fts2, otherwise
    where
    fts2 = filter (~= NIL)
                  [delete nm ft|ft<-fts]
```

The expression `commit nm ft` is the feature tree whose products are the products of the feature tree `ft` which contain the feature with name `nm`.

```
commit :: name -> tree -> tree
commit nm NIL = NIL
commit nm ft = ft2, if b
             = NIL, otherwise
               where
               (ft2,b) = commit2 nm ft

commit2 :: name -> tree -> (tree,bool)
commit2 nm NIL = (NIL,False)
commit2 nm (MandOpt n ms os)
  = (MandOpt n ms os, True), if n=nm
  = (MandOpt n ms3 os, True),
                       if or (map snd ms2)
  = (MandOpt n (ms++os4) os3, True),
                           if os4 ~= []
  = (MandOpt n ms os, False),  otherwise
    where
    ms2 = [commit2 nm m|m<-ms]
    os2 = [commit2 nm o|o<-os]
    ms3 = [m|(m,b)<-ms2]
    os3 = [ft|(ft,b)<-os2; ~b]
    os4 = [ft|(ft,b)<-os2; b]
commit2 nm (Xor n fts)
  = (Xor n fts, True), if n=nm
  = (MandOpt n ms [], True), if #ms = 1
  = (Xor n ms, True), if #ms>=1
  = (Xor n fts, False), otherwise
    where
    fts2 = [commit2 nm ft|ft<-fts]
    ms = [ft|(ft,b)<-fts2; b]
commit2 nm (Or n fts)
  = (Or n fts, True), if n=nm
  = (MandOpt n ms os, True), if ms ~= []
  = (Or n fts, False), otherwise
    where
    fts2 = [commit2 nm ft|ft<-fts]
    os = [ft|(ft,b)<-fts2; ~b]
    ms = [ft|(ft,b)<-fts2; b]
```

The expression `ElimConstr` takes a feature model as argument, and returns a feature tree with the same semantics

```
elimConstr :: feature_model -> tree
elimConstr (ft, Requires a b : cs)
  = elimConstr (Xor "" fts, cs),if #fts>1
  = elimConstr (MandOpt "" fts [], cs),
                                 if #fts=1
  = NIL, otherwise
    where
    fts = filter (~=NIL) [delete a
              (delete b ft), commit b ft]
elimConstr (ft, Excludes a b : cs)
  = elimConstr (Xor "" fts, cs),if #fts>1
  = elimConstr (MandOpt "" fts [], cs),
                                 if #fts=1
  = NIL, otherwise
    where
    fts = filter (~=NIL) [delete b ft,
                  delete a (commit b ft)]
elimConstr (ft,[]) = ft
```

The function `nrProds` computes the number of products of a feature tree

```
nrProds :: feature_tree -> num
nrProds NIL = 0
nrProds (MandOpt nm ms os)
  = product (map nrProds ms) *
    product (map (+1) (map nrProds os))
nrProds (Xor nm fts)
  = sum (map nrProds fts)
nrProds (Or nm fts)
  = product (map(+1) (map nrProds fts))-1
```