Analysis of Feature Models using Generalised 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

Abstract

This paper introduces the concept of generalised feature trees, which are feature trees where features can have multiple occurrences. It is shown how an important class of feature models can be transformed into generalised feature trees. We present algorithms which, after transforming a feature model to a generalised feature tree, compute properties of the corresponding software product line. We discuss the computational complexity of these algorithms and provide executable specifications in the functional programming language Miranda.

1. Introduction

Feature models are used to specify the variability of software product lines [1,2]. To calculate properties of software product lines which are specified by feature models, such as the existence of products, a number of approaches exist in the literature where feature models are mapped to other data structures: Benavides et al. [3] use Constraint Satisfaction Problems, Batory [4] uses Logic Truth Maintenance Systems and Satisfiability Solvers, and Czarnecki and Kim [5] use Binary Decision Diagrams.

The decision problem to determine whether a feature model has products is, in general, NP-complete. The mappings to Constraint Satisfaction Problems and Logic Truth Maintenance Systems can be performed in polynomial time, but the resulting problem is also NP-complete. Although with Binary Decision Diagrams the problem only requires constant time, the mapping from feature models to Binary Decision Diagrams takes exponential time in the worst case.

In a previous paper [6] we have shown how feature models which consist of a feature tree and additional constraints can be transformed into trees. Although this transformation takes exponential time in the worst case as well, it is feasible when the number of constraints is small. The resulting trees are more general than feature trees, since features may have multiple occurrences. In this paper we study a special subset of those trees, called *generalised feature trees*, and show how they can be used to compute properties of the corresponding software product lines.

In the next section we briefly describe the feature models we consider in this paper. In section 3 we introduce the concept of generalised feature tree and describe algorithms which deal with commitment to a feature and deletion of a feature of a GFT. In section 4 we describe how a large class of feature models can be mapped to equivalent GFTs. In section 5 we show how this mapping can be used for the analysis of feature models. In section 6 we present an example and in section 7 we discuss the computational complexity of our approach. Throughout the paper, we present executable specifications of all algorithms in the functional programming language Miranda [8].

2. Feature models

The feature models we consider in this paper consist of a feature tree and a set of constraints. A feature tree is a tree which can have three kinds of nodes: MandOpt nodes, Or nodes and Xor nodes.

A MandOpt node has two sets of child nodes, called mandatory and optional nodes respectively. Or nodes and Xor nodes have 2 or more child nodes. A leaf of the tree is a MandOpt node without children. Just for the

ease of writing concise algorithms, we assume the existence of a special feature tree NIL, which has no nodes. Each node of a tree has a feature, which is just a list of characters. All nodes in a feature tree have different features, and NIL does not occur as subtree of any feature tree. A product is a set of features. A constraint maps products to Boolean values; in our prototype implementation the constraints are restricted to constraints of the forms "A requires B" and "A excludes B".

In Miranda, these type definitions are as follows:

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

The semantics of a feature model is a set of products [7]; it consists of those products which satisfy the constraints from the tree as well as the explicit constraints.

A product satisfies the constraints from the tree if:

- All its features are features of a node in the tree.
- It contains the feature of the root of the tree.
- For each feature of a node n in the product: if n is not the root, then the product contains also the feature of the parent node of n.
- For each feature of a MandOpt node in the product, the product also contains all features of its mandatory child nodes.
- For each feature of an Or node in the product, the product also contains one or more of the features of its child nodes.
- For each feature of an Xor node in the product, the product also contains exactly one of the features of its child nodes.

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

3. Generalised feature trees

Features in a feature tree are, albeit implicitly, required to be all distinct. In the generalisation of feature trees we consider in this paper, this requirement is somewhat relaxed. We define a generalised feature tre (GFT) to be a feature tree whose features, instead of being required to be all distinct, satisfy the following two restrictions:

- Restriction 1: when two nodes of a GFT have the same feature, they belong to different subtrees of an Xor node.
- Restriction 2: for each node of a GFT, all subtrees have disjoint semantics.

Before motivating both restrictions, we will first define the semantics of a GFT. As in the previous section, this semantics is a set of products. The definition of the previous section relied on the 1-1 correspondence between nodes and features, which does not exist here. Therefore, we first define a set of sets of nodes, instead of a set of products as in the previous section. This set of sets of nodes contains each set of nodes which satisfies

- It contains the root of the GFT.
- For each node in the set except the root, the set also contains its parent node.
- For each MandOpt node in the set, the set also contains all its mandatory child nodes.
- For each Or node in the set, the set also contains one or more of its child nodes.
- For each Xor node in the set, the set also contains exactly one of its child nodes.

The semantics of the GFT is now defined as the set of products which is obtained from this set of sets of nodes when each node is replaced by its feature. Where each feature tree is a GFT, it is seen that this definition coincides with the definition of the previous section when the GFT is a feature tree.

Although a GFT may contain multiple occurrences of a feature, we do not want multiple occurrences of features in products. This is the motivation of the first restriction above; it prevents multiple occurrences of features in products.

In a feature tree, different subtrees of a node do not contain equal features; this means that the semantics of these subtrees are disjoint. For a GFT, different subtrees of a node may contain equal features; however, we still want the semantics of these subtrees to be disjoint, as is expressed by the second restiction above. The reason for this is that computations might become inefficient otherwise. For instance, consider a GFT whose root node is an Xor node which has two subtrees and suppose these subtrees have N and M products respectively. When we know that these sets of products are disjoint we can conclude that the total number of products is N+M. Otherwise, we have to single out common products from both sets.

An important property of GFTs which is not valid for feature trees is that for each set of products there exists a GFT. Given a set of products, a corresponding GFT can be constructed as an Xor root node with

subtrees for each product; each subtree corresponds to a single product.

In the remainder of this section we present two algorithms, which deal with commitment to a feature and deletion of a feature of a GFT, respectively. These algorithms are generalisations of algorithms for feature trees which are given in [6].

The first algorithm computes, given a GFT T and a feature F, the GFT T(+F), whose products are precisely those products of T which contain F. The algorithm transforms T into T(+F) as follows:

1. If T does not contain F, T(+F) is NIL, else GOTO 2
2. If F is the feature of the root node of T, T(+F) is T, else GOTO 3
3. Execute 4, 5 or 6, depending on whether the root of T is a MandOpt node, an Or node or an Xor Node
4. Determine the unique subtree S which contains F, determine S(+F) recursively, and replace S by S(+F). If the root node of S was an optional node, make the root of S(+F) a mandatory node.
5. Determine the unique subtree S which contains F, determine S(+F) recursively, and replace T by a MandOpt node, with the same feature as T, which has S(+F) as mandatory subtree and all other subtrees of T as optional subtrees.
6. Determine the subtrees $S_1,..,S_n$ which contain F, determine $S_1(+F),..,S_n(+F)$ recursively, and replace $S_1,..,S_n$ by $S_1(+F),..,S_n(+F)$. Delete all other subtrees. If n=1, make the root node of T a MandOpt node, and its subtree a mandatory subtree.

The second algorithm computes, given a GFT T and a feature F, the GFT T(–F) whose products are precisely those products of T which do not contain F. The algorithm transforms T into T(–F) as follows:

1. If T does not contain F, T(–F) is T, else GOTO 2
2. If F is the feature of the root node of T, T(–F) is NIL, else GOTO 3
3. Execute 4, 5 or 6, depending on whether the root of T is a MandOpt node, an Or node or an Xor Node
4. Determine the unique subtree S which contains F and determine S(–F) recursively. If S is mandatory and S(–F) = NIL, then T(–F) is NIL. IF S is optional and S(–F) = NIL delete S from T. If S(–F) ≠ NIL then replace S by S(–F).
5. Determine the unique subtree S which contains F, determine S(–F) recursively. If S(–F) ≠ NIL, replace S by S(–F). If S(–F) = NIL, delete S. If T has only 1 subtree left, make its root node a MandOpt node, and its child a mandatory child.
6. Determine the subtrees $S_1,..,S_n$ which contain F and determine $S_1(–F),..,S_n(–F)$ recursively, and delete all other subtrees. For i=1,..,n, if $S_i(–F)$ = NIL, delete S_i, otherwise replace S_i by $S_i(–F)$. If T has no subtrees left, then T(–F) is NIL If T has only 1 subtree left, make its root node a MandOpt node, and its child a mandatory child.

In [6] we gave an implementation in Miranda of functions with type definitions

```
commit :: feature -> tree -> tree
delete :: feature -> tree -> tree
```

These functions, originally given for feature trees, need no modification to be applicable to GFTs as well. The function `commit` takes a feature F and a GFT T as arguments, and returns T(+F), as defined above. Likewise, the function `delete` returns T(–F).

4. From feature models to generalised feature trees

In [6] we showed how a feature model which consists of a feature tree and Requires and/or Excludes constraints can be transformed into a tree with the same semantics. Here we will generalise this method to general constraints, and show that the resulting tree is a GFT. Suppose we are given a feature tree T and a constraint C, which is a mapping from P, the set of all products with features of T to the Boolean values {True,False}. Find a partition of P such that C is constant on each part. For each part on which C is True, find a corresponding GFT, using the algorithms of the previous section. Finally obtain the GFT whose root node is an Xor node and which has the GFTs just found as child nodes. As an example, consider the constraint C to be "A requires B". Partition P into {P(+B), P(–A–B), P(+A–B). Here "+A" and "–A" denote restriction to products where A is present resp. absent. C is True on P(+B) and P(–A–B) and C is False on P(+A–B). GFTs for P(+B) and P(–A–B) are T(+B) and T(–A–B). The resulting GFT has an Xor root node and T(+B) and T(–A–B) as subtrees. Analogously, if C is "A excludes B", the new GFT has an Xor root node and T(–B) and T(–A+B) as subtrees.

The resulting trees are indeed GFTs. Restriction 1 is satisfied because all generated subtrees have the same Xor root node as parent node. Restriction 2 is satisfied because the semantics of the generated subtrees are the parts of a partition of P, and therefore have no common features.

In [6] we gave an implementation in Miranda of a function with type definition

```
elimConstr :: feature_model -> tree
```

The argument of this function is a feature model which consists of a feature tree and constraints of the forms "A requires B" and "A excludes B"; the function returns a corresponding GFT.

5. Analysis of feature models

In this section we show how the algorithms of the preceding section can be used to analyse feature models which consist of a feature tree and a number of constraints. In this implementation the constraints are restricted to be of the forms the function `elimConstr` can handle; these are the forms "A requires B" and "A excludes B",but other forms might be included as well, as described in the previous section.

Starting point of the analysis is a feature model consisting of the feature tree `f_tree` and the list of constraints `constraints`. The first step of the analysis is the computation of an equivalent GFT `gft`:

```
gft :: tree
gft = elimConstr (f_tree, constraints)
```

The function `elimConstr` here can be used if all constraints are of the forms "A requires B" and "A excludes B"; otherwise, the procedure described in the previous section should be followed.

In the remainder of this section we describe the computation of a number of properties of the specified software product line.

Existence of products

The feature model has products if and only if `gft` is not equal to NIL:

```
has_products :: bool
has_products = gft ~= NIL
```

Dead features

The dead features of the feature model are the features which occur in features but do not occur in `gft`:

```
dead_features :: [feature]
dead_features
   = features f_tree -- features gft
```

Here the function `features` computes a list of all features of a GFT:

```
features :: tree -> [feature]
features (MandOpt f ms os)
   = f : concat (map features (ms++os))
features (Or f fts)
   = f : concat (map features fts)
features (Xor f fts)
   = f : concat (map features fts)
```

Number of products

The number of products of the feature model is

```
nr_products :: num
nr_products = nrProds gft
```

where the function `nrProds` is given by

```
nrProds :: 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
```

List of all products

A list of all products of the feature model is

```
list_of_products :: [[feature]]
list_of_products = products gft
```

where the function `products` computes a list of products of a GFT:

```
products :: tree -> [[feature]]
products (MandOpt x ms os)
 = map(x:)(f(map products ms ++
             map([]:)(map products os)))
   where
   f [] =  [[]]
   f(xs:xss) = [u++v|u<-xs;v<-f xss]
products (Xor x fts)
 = map(x:)(foldl(++)[](map products fts))
products (Or x fts)
 = map(x:)(f(map products fts)--[[]])
   where
   f [] = [[]]
   f(xs:xss) = [u++v|u<-([]:xs);v<-f xss]
```

Products which contain a given set of features

A GFT whose products are precisely those products of `gft` which contain all features from a list `required_features` is:

```
gft2 :: tree
gft2 = gft_req_fts required_features gft
```

where the function `gft_req_fts` is defined by:

```
gft_req_fts :: [feature] -> tree -> tree
gft_req_fts [] t = t
gft_req_fts (f:fs) t
       = commit f (gft_req_fts fs t)
```

Minimal set of conflicting constraints

A set of constraints is in conflict with a feature tree if the feature model consisting of this tree and these constraints has no products, i.e when `gft` evaluates to NIL. A user, confronted with such a conflict, may want some explanation of this. A solution might be to provide the user with a smallest minimal set of constraints that conflict with the feature tree. A minimal set of constraints is a set which contains conflicting constraints, but has no proper subset whose constraints also conflict. A smallest minimal set of conflicting constraints can be computed by

```
confl_constr :: [constraint]
confl_constr = smsocc(f_tree,constraints)
```

where the function `smsocc` (smallest minimal set of conflicting constraints) is given by:

```
smsocc :: feature_model -> [constraint]
smsocc (t,[]) = []
smsocc (t,c:cs)
  = [c], if t2 = NIL
  = [], if set1 = []
  = c:set1, if set2 = [] \/ #set2>#set1
  = set2, otherwise
    where
    t2 = elimConstr (t,[c])
    set1 = smsocc (t2,cs)
    set2 = smsocc (t,cs)
```

This function, given the original feature model as argument, returns a list with a minimal set of conflicting constraints if `gft` equals NIL; otherwise it returns the empty list.

Explanation of dead feature

If `dead_features`, the list of dead features, is non-empty and contains the feature `dead_feature`, the user might want explanation why this feature is dead. As above, this explanation is a minimal set of constraints which causes the feature to be dead. It is given by

```
expl_dead_ft :: [constraint]
expl_dead_ft
  = explain (f_tree,constraints)
            dead_feature
```

where the function `explain` is given by

```
explain :: feature_model
           -> feature -> [constraint]
explain (t,cs) f
  = smsocc (t2,cs), if t2 ~= NIL
  = [], otherwise
    where
    t2 = commit f t
```

The arguments of this function are the original feature model and a feature from the list `dead_features`. It returns a minimal set of constraints which causes the feature to be dead. If the feature does not belong to `dead_features`, the empty list is returned.

6. Example

As an example, consider the feature tree T in Figure 1, which is adapted from [9].

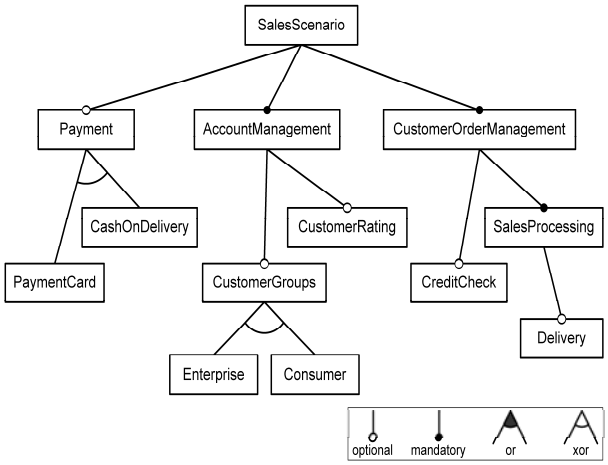

Figure 1. Example feature tree T

In Miranda, the definition of `f_tree` is

```
f_tree = MandOpt "SalesScenario"
         [n2,n3] [n1]
n1 = Xor "Payment" [n4,n5]
n2 = MandOpt "AccountManagement"
     [] [n6,n7]
n3 = MandOpt "CustomerOrderManagement"
     [n9] [n8]
n4 = MandOpt "PaymentCard" [] []
n5 = MandOpt "CashOnDelivery" [] []
n6 = Xor "CustomerGroups" [n10,n11]
n7 = MandOpt "CustomerRating" [] []
n8 = MandOpt "CreditCheck" [] []
n9 = MandOpt "SalesProcessing" [] [n12]
n10 = MandOpt "Enterprise" [] []
n11 = MandOpt "Consumer" [] []
n12 = MandOpt "Delivery" [] []
```

Our example feature model consists of this feature tree T and the 2 constraints: "CashOnDelivery excludes Consumer" and "Enterprise requires Consumer". So the list `constraints` is given by

```
constraints = [c1,c2]
c1 = Excludes "CashOnDelivery" "Consumer"
c2 = Requires "Enterprise" "Consumer"
```

The GFT `gft` is given in Figures 2, 3 and 4. It could have been computed with the function `elimConstr`, since both constraints are of the forms "A requires B" and "A excludes B"; however, we will illustrate the method to derive it which was given in section 4. If "Consumer" is present in a product, the constraints are satisfied iff "CashOnDelivery" is not present. If "Consumer" is absent in a product, the constraints are satisfied iff "Enterprise" is also absent. So the set of products P can be partitioned in such a way that the parts P(+Consumer–CashOnDelivery) and P(–Consumer–Enterprise) consist of the products which satisfy the constraints. Therefore, the equivalent GFT is given by a new Xor node which has T(+Consumer–CashOnDelivery) and T(–Consumer–Enterprise) as subtrees.

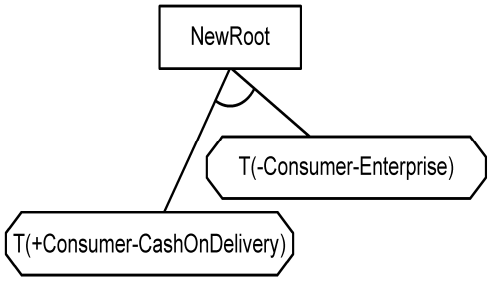

Figure 2. generalised feature tree `gft`, toplevel

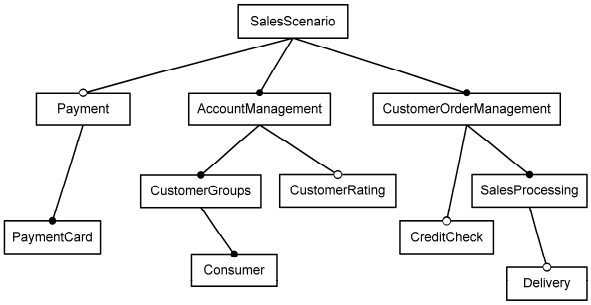

Figure 3. T(+Consumer–CashOnDelivery)

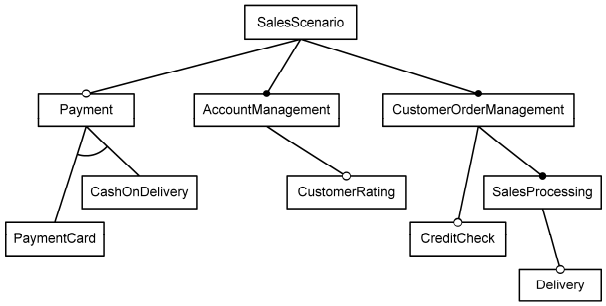

Figure 4. T(–Consumer–Enterprise)

The analysis of this example feature model proceeds as follows:

- `has_products` evaluates to `True`.
- `dead_features` evaluates to `["Enterprise"]`, showing that the feature "Enterprise"is dead.
- `nr_products` evaluates to `40`.
- `list_of_products` evaluates to a list of the 40 products (not shown for brevity).
- if `required_features` is defined as a list of features then `gft2` evaluates to a GFT which can be analyzed in the same manner.
- if `dead_feature` is defined to be `"Enterprise"` then `expl_dead_ft` evaluates to `[Requires "Enterprise" "Consumer"]` showing that the second constraint is on its own responsible for the deadness of `"Enterprise"`.

Now suppose that an extra constraint "SalesProcessing requires Enterprise" is added:

```
constraints = [c1,c2,c3]
c3 = Requires "SalesProcessing"
              "Enterprise"
```

Now `has_products` evaluates to `False` and `confl_constr` evaluates to `[Requires "Enterprise" "Consumer", Requires "Sales Processing" "Enterprise"]` which shows that the second and third constraints together form a smallest minimal set of constraints that conflict with the feature tree.

7. Computational Complexity

We have shown in [6] that the decision problem whether a feature model which is given by a feature tree and a set of constraints is NP-complete. Therefore, we cannot hope that the analysis of such a feature model can be performed in polynomial time in the worst case. Indeed, the construction of the GFT for the feature model takes a time which is exponential in the number

of constraints in the worst case. Also the algorithm for the computation of a minimal set of constraints which conflict with the feature tree and the algorithm which computes the minimal set of constraints which cause a feature to be dead are exponential in the number of constraints. However, once the GFT has been constructed, the algorithms for the existence of products, the number of products and the list of dead features are linear in the size of the GFT.

In the special case where the number of explicit constraints is 0, the intended GFT is the feature tree without modification. Then `has_products` belongs to O(1), and `nr_products` belongs to O(N). This certainly outperformes the other analysis methods mentioned in the introduction, as these methods require a transformation of the feature tree to another data structure; in the case of Binary Decision Diagrams this transformation requires even exponential time in the worst case. We expect that our method is more efficient than the other methods also in the case where the number of constraints is small. For instance, it has been shown in [6] that `nr_products` belongs to $O(N*2^M)$, where N is the number of features and M is the number of constraints. A more detailed comparison is planned as a future work.

8. Conclusion

We have introduced the concept of generalised feature trees and have shown how they can be used to analyse feature models which consist of a feature trees and additional constraints. Detailed algorithms have been given in the functional programming language Miranda. The algorithms are efficient when the constraints are a small number of "requires" and/or "excludes" constraints

9. Acknowledgments

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

10. 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. van den Broek, I. Galvão and J. Noppen, "Elimination of Constraints from Feature Trees" in: Steffen Thiel and Klaus Pohl (Eds.), Proceedings of the 12th International Software Product Line Conference, Second Volume, Lero International Science Centre, University of Limerick, Ireland, ISBN 978-1-905952-06-9, 2008, pp. 227-232.

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

[8] 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.

[9] H. Morganho, J.P. Pimentão, R. Ribeiro, C. Pohl, A. Rummler, C. Schwanninger and L. Fiege, "Description of Feasible Industrial Case Studies", Deliverable 5.1, AMPLE Project, http://www.ample-project.net/ (2007).