First Workshop on

Analyses of Software Product Lines

Limerick, Ireland 2008

Finding Contradictions in Feature Models

Adithya Hemakumar
Dept. Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
hemakuma@ece.utexas.edu

Abstract

A feature model defines each product in a product-line by a unique combination of features. Feature compatibilities are expressed as constraints in feature models and may be contradictory. We suggest a run-time approach to expose contradictions in feature models when they are uncovered. However, the emphasis of this paper is to explore the possibility of finding contradictions statically using model checking and an incremental consistency algorithm.

1. Introduction

A *feature model* is a common way to express the products of a *software product line (SPL)*. A *feature* is an increment in product development, and no two products in an SPL have the same combination of features. A feature model can be formally defined as a *context free grammar (CFG)* with constraints; the tokens of the grammar are primitive features and the constraints eliminate nonsensical combinations of features that the grammar would otherwise admit [4][5][16].

A feature model can be mapped to a propositional formula, where each feature is equated with a distinct boolean variable and the formula encodes the constraints of feature compatibility [5][3][6][21][30]. A variable has the value **true** if its corresponding feature is present in a product; it is **false** otherwise. A truth assignment to the variables that satisfies the formula defines a product in the feature model's SPL, and each product has a unique truth assignment.

Behind every feature model lurks the possibility of contradictions. Consider the following elementary feature model **M** whose CFG is:

```
M : [A] [B] C D;
```

All products of **M** have features **C** and **D**, and optionally features **A** and **B**. Suppose the following (obviously nonsensical) feature compatibility constraints are added to **M**:

```
A implies B;
B implies not A;                                   (1)
```

Observe that a contradiction arises if feature **A** is selected: **B** is selected by enforcing the first constraint, and then **A** is deselected by the second. Selecting **A** implies its deselection is clearly both an error and contradiction in the model. Such contradictions reveal *dead features* — features that are present in no product [5][6]. For example, **A** is present in no product of **M**. Benavides et al. presented an analysis for finding *unconditionally* dead features in feature models [6], i.e., features that are dead without preconditions. In this paper, we show that contradictions can arise in more general ways: dead features can arise *conditionally*. Consider model **G**:

```
G : [A] [B] [C] D ;
```

All products of **G** have feature **D**, and optionally features **A**, **B**, **C**. The feature compatibility constraints for **G** are:

```
(C and A) implies B;
B implies not A;                                   (2)
```

Note that model **G** is a generalization of **M** because all products of **M** are products of **G**. Further, **A** is *not* a dead feature in **G**: there is a product of **G** with feature **A** (namely product **AD**). However, if feature **C** is selected, then model **G** simplifies to model **M**. The contradiction is exposed in **G** when features **C** and then **A** are selected in this order. Stated differently, feature **A** is dead whenever feature **C** is selected. No analysis that we are aware (including [6]) uncovers the conditions under which **A** is dead or when model **G** is contradictory.

Like any specification, designers need to be alerted to such errors, and feature models are no exceptions. Constraints like **(1)** are clearly wrong, but when extra conditions are added (as in **(2)**) they become next to impossible to spot manually.

Finding contradictions (i.e., conditions under which features are dead) in a feature model is a challenging problem. In this paper, we suggest a solution to expose contradictions at run-time by noting when different constraint propagation algorithms have different outputs. Our emphasis is to explore the difficulty of finding contradictions statically using model checking and an incremental consistency algorithm.

2. A Run-Time Solution and Perspective

Our work is at the confluence of a number of different research threads in SPLs. *Binary Decision Diagrams (BDDs)* are a common way to represent propositional formu-

las in product specification tools [3][30]. Among the analyses that can be performed by BDDs is constraint propagation. Given a set of features that must be in a target product, analyses can infer the selection and deselection of other features (obeying the compatibility constraints in a feature model). For example, a BDD can infer that *no* product in **M** of the Introduction uses feature **A**. A GUI front-end uses this information to preclude users from selecting **A**, thereby avoiding the contradiction of specifying a product in **M** with **A**. (In the AI configuration community [1][2][10], this property is called *backtrack-free* — a configurator should not allow a user to select a feature that leads to invalid configurations [27][28][30]). It is well-known that the BDD inferencing of constraints is *complete* — all possible inferences are made.

An alternative to BDDs is the *Boolean Constraint Propagation (BCP)* algorithm [4]. BCP is a classical AI algorithm used in logic truth maintenance systems and works by iteratively applying rules (constraints) to infer the truth values of variables [11]. Unlike BDDs, BCP is *incomplete*. That is, BCP cannot infer all facts. In the case of model **M**, the following can happen: the GUI front-end allows users to select feature **A** (because BCP was unable to infer that **A** cannot be present in a product of **M**) and when **A** is chosen, constraint propagation reveals the contradiction in **M**.

From a product-specification-tool perspective, BDDs are preferred over BCP as they preclude users from specifying a set of features that will lead to a contradiction. *But this does not eliminate the error in model* **M**. Using BDDs, users or designers will discover that they can *never* select feature **A** and this is a tip-off to an error. In contrast, BCP makes the error of **M** visible: it explicitly reports that **M** is contradictory by showing two different inference chains that lead to opposite conclusions. *But this requires users to select a specific sequence of features to expose the error.*

Feature incompatibility is expressed by unsatisfiability — two features **A** and **B** are incompatible w.r.t. predicate `P(A,B)` if `P(true,true)=false`. That is, `P(true,true) =false` means features **A** and **B** cannot both be present in the same product. Modelling errors — where two different inference chains lead to contradictory results — are *also* expressed by unsatisfiability: there is one inference chain where `P(true,true)=false` and of course there is another chain where `P(true,true)=true`. Feature modelling tools that use BDDs to propagate constraints do not distinguish feature incompatibilities from contradiction errors; tools based on BCP do. Thus, BCP algorithms can be used to identify contradiction errors in models.

The above suggests a simple, solution for finding contradictions in feature models: Product configuration tools should use BDDs to side-step contradictions. But they should also use BCP algorithms to also propagate constraints. Under normal circumstances, BDDs and BCP will produce identical outputs when propagating constraints. However, when BDDs assign values to variables that BCP does not, a contradiction has been exposed. Such findings can be quietly reported by the tool to model designers for subsequent examination and model repairs.

Ideally, however, we want to discover contradictions not at run-time (when users are selecting features to define products), but through static analysis. In the following sections, we explore several approaches, using model checking and an incremental consistency algorithm, that exposes contradictions statically. We begin by reviewing the details of feature models and the BCP algorithm.

3. Feature Models and the BCP Algorithm

A *feature diagram* is a common way to depict a feature model [9][16][17]. It is an and-or tree, where children of a node can be optional or mandatory. Terminals are primitive features and non-terminals are compound features. Constraints on selecting a particular number of children (choose exactly one child or choose one or more) and cross-tree constraints (predicates that relate features of different subtrees) can be declared. Figure 1a shows a feature diagram of model **M** from Section 1. using common notations that are defined in [9]. Cross-tree constraints are not depicted.

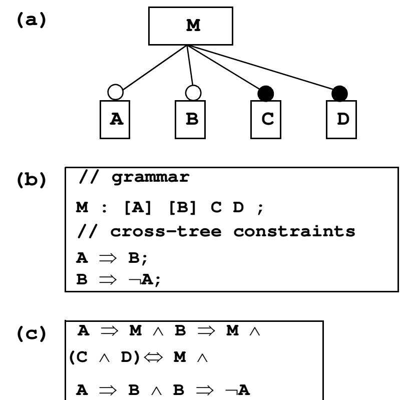


Figure 1 Feature Diagrams

As mentioned in Section 1., another representation of a feature diagram is a CFG with cross-tree constraints. Figure 1b shows this representation for model **M**. Simple rules translate a CFG into a propositional formula, and the cross-tree constraints are conjoined onto this formula (Figure 1c) [4]. The resulting formula can then be translated into *conjunctive normal form (CNF)* for subsequent analysis.

The BCP algorithm is simple [11]. A CNF clause is *unit open* if all but one of its terms (i.e., a variable or its negation) is `false`. The BCP algorithm uses unit open clauses to infer the value of an unassigned term. For example, if **x** and **y** are `true` in the unit open clause $(\neg x \lor \neg y \lor z)$ then the BCP algorithm concludes **z** must be `true`. When a variable is

assigned a value, every CNF clause of a feature model's formula is examined and each unit open clause is pushed on a stack.

The BCP algorithm is a loop: the stack is popped, the popped clause is checked if it remains unit open, and if so a variable assignment is inferred (triggering more clauses to be pushed onto the stack). The BCP loop terminates when the stack is empty. By remembering the sequence of inferences that are made, explanations of variable assignments can be presented to users in the form of a proof [16].

From a tool perspective, a product of a product-line is declaratively specified by selecting its features one feature at a time. After each feature selection, BCP is invoked to propagate constraints. It is during constraint propagation that model contradictions are discovered.

A CNF clause is *violated* when all of its terms are false. When BCP encounters a violated clause, a model contradiction is announced. The incompleteness of the BCP algorithm is evident from its description: using model **`M`** of Figure 1, *only* when feature **`A`** is selected will the contradiction of **`M`** reveal itself.

The general problem of finding model contradictions is to find a sequence of k feature selections (where n is the total number of features and $0<k\leq n$) that reveal two inference chains that reach opposing conclusions. A feature model is *contradiction free* if there are no contradictions for all k, $0<k\leq n$. In the following sections, we examine two algorithms to find contradiction errors statically.

- **Spin**: Contradictions can be found by model checking. The goal is to prove that error states that correspond to contradictions cannot be reached for a given feature model. Spin is a model checker that interprets a Promela program. The system (BCP + feature model) is represented by a Promela program and Spin performs an exhaustive search on its state space.
- **Incremental Consistency**: A contradiction is a result of constraint propagation of the kth feature selection given a sequence of *k-1* previous selections. By induction, we incrementally prove via enumeration that a model is consistent for increasing values of k, for $0<k\leq n$.

4. Spin (Simple Promela INtrepreter)

Specifying a product using a feature modelling tool alternates between two phases: the user selection of a feature and the propagation of constraints. This process can be visualized as a state machine, where each phase repeats over time (Figure 2).

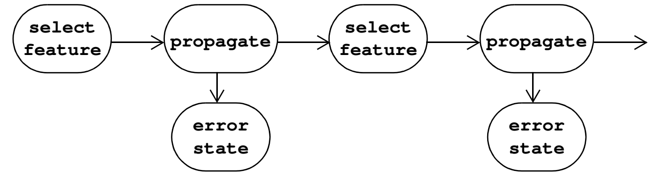


Figure 2 State Machine of Feature Selection and Constraint Propagation

Each state of a state machine represents a unique value assignment to the set of variables (features) of the given feature model. In BCP, variables can assume one of three values — **`unknown`**, **`true`**, or **`false`**. All features (with the exception of the root of the grammar) start with **`unknown`** as their initial value (the root is assigned **`true`**). Upon each feature selection, the system propagates constraints to select and deselect other features obeying the constraints of the feature model. When no further inferences can be made, the cycle repeats by selecting the next feature.

Special states, called *error states*, arise during constraint propagation when a CNF formula is violated (thus implying a model contradiction). A model checker is a general-purpose tool for traversing a state machine to determine if particular states (or conditions on states) are reachable. Finding a sequence of user feature selections that results in a contradiction can be formulated as a reachability problem.

We used Spin for our work [25][14]. Promela is a modelling language for concurrent processes. It was designed for verification and its programs can be directly model checked by Spin. The variables and constraints of the specified feature model are represented as propositional formulas. A Promela-translator converts these formulas into a Promela program. User selections are modelled as an exhaustive sequential selection of user-visible features.

Spin provides three ways to flag errors: we used assertions to define error states [25]. Spin's verification procedure is based on a depth-first search of the state space. It offers two modes of operation:

- **Exhaustive search**: The entire state space is examined. This is the default mode of operation.
- **Bitstate hashing**: For large problem sizes, an exhaustive depth-first graph traversal method is not possible due to the bound placed by the size of the memory. In such cases, a high-coverage approximation of the exhaustive runs can be performed with the available memory using bitstate hashing [15]. This technique strikes a compromise between the number of states explored and the amount of the memory used. If the problem size is more than the size of available memory, Spin covers only a fraction of the state space.

Figure 3 shows a Promela file that implements the BCP algorithm for the propositional constraints of model **M** in Section 1.. One of three values — **T** (**true**), **F** (**false**), **U** (**unknown**) — is assigned to each selectable variable/feature. Each assignment to the set of variables defines a unique state of the machine. The initial assignment of values is indicated by **(*)** in Figure 3, where features **A** and **B** are assigned the value **U**.

When BCP infers the value of a feature, a state transition occurs. Transitions are represented by **if** statements in Promela. The two **if** statements following the **propagate:** label are the BCP actions for inferring features for two CNF clauses, which correspond to the two constraints of **M**: **A**⇒**B** and **B**⇒¬**A**. Each CNF clause has two terms. If one is **F** and the other is **U**, the value of the other feature is inferred. Only after all constraints have been propagated, is another feature selected. Note: after each **if** statement is an assertion, which if violated, indicates a model contradiction error.

The **if** statement following the "**Select Feature Phase**" comment selects a previously unselected feature (i.e., whose value is **U)**. If there are no **U**-valued features, all features/variables have been assigned **T** or **F** values, the search backtracks. Only when the entire search space (i.e., all possible sequences of feature selections) has been examined does the search terminate.

We built a tool that translates a feature model (CNF grammar + constraints) into a Promela program. The program is then compiled; its execution explores the state machine of the feature model. We used a number of different feature models, which we ourselves previously wrote in building SPLs for different domains, or that others had written using our tools. We deliberately introduced contradictions in some models to test our tools. These models are summarized in Figure 4. Model complexity is indicated by the number of features and CNF clauses.

```
mtype = {T, F, U};

init {
    mtype A=U, B=U;                              (*)

    do: // loop till you finish selecting all features

        /********** Propagate Phase **********/
        propagate:
        if // Promela code for constraint A ⇒ B
         ::(A == U && B == F) -> A = F; goto propagate;
         ::(B == U && A == T) -> B = T; goto propagate;
         ::else -> skip;
        fi;
        assert(!(A == T && B == F )); //error state
        if // Promela code for constraint B ⇒ ¬A
         ::(B == U && A == T) -> B = F; goto propagate;
         ::(A == U && B == T) -> A = F; goto propagate;
         ::else -> skip;
        fi;
        assert(!(B == T && A == T )); //error state
        /********** Select Feature Phase **********/
        if
         ::A==U -> A = T; printf("choose feature A\n");
         ::B==U -> B = T; printf("choose feature B\n");
         ::else -> break;
        fi;
    od;
}
```

Figure 3 Portion of a Promela File for Model **M**

Figure 5 shows the execution time for exhaustive search and bitstate hashing (and its coverage factor) of Spin for these models. We ran all of our experiments (in this section and the next) on an Intel Pentium IV processor with a 1GB RAM. For small models (models having less than 10 features) Spin completes the check within seconds. For larger models, system memory is exhausted quickly. "∞" indicates models for which Spin never terminated. Bitstate hashing helps restrict memory consumption but the search is not exhaustive. As bitstate hashing does not cover the entire state space, error states might not be found.

The models that we were able to analyze completely with Spin either had no inconsistencies, or caught the errors which we deliberately injected. (From our prior experience in building models, we knew that some of our models had inconsistencies, but those errors had been found and corrected prior to this research. However, we did not know if our models were contradiction free).

Our experience with Spin was mixed. While it was successful, it was clear that we had to reduce the size of the state space, as Spin could not provide us with certifications that all of our models were contradiction free. We also realized that in a sequence of $k+1$ feature selections, the order in which the first k features are selected does not matter. That

Model Name	# of Features	# of CNF Clauses	Brief Description of the Model's SPL
BerkeleyDB	55	185	BerkeleyDB [18]
Folutest	13	66	a notepad application
Freeman	3	17	a scalar vector graphics application
GG4-model	15	140	an elaborated graph product line
GPL	17	188	graph product line [20]
Notepad	20	155	a notepad application
SVGMap	19	52	a SVG map application
TightVNC	21	83	desktop sharing application
Violet	64	341	image processing application
apl	12	47	error-injected model
long	12	17	error-injected model

Figure 4 Different Feature Models Used In Our Experiments

Model Name	Exhaustive (secs)	Bitstate Hashing (secs)	BitState Hashing Coverage (%)
BerkeleyDB	∞	106.4	14
Folutest	3.5	3.3	100
Freeman	1.2	1.3	100
GG4-model	8.6	6.7	100
GPL	∞	22.8	85
Notepad	∞	67.3	21
SVGMapApp	∞	52.1	24
TightVNC	∞	95.7	14
Violet	∞	102.4	19
apl	4.8	1.7	100
long	1.6	1.6	100

Figure 5 Execution Time of Spin for Different Feature Models

is, the value assignments to the feature/variables after the first *k* features are selected (and constraints propagated) are invariant to the order in which these features are chosen [12]. This observation allowed us to reduce the size of the state-space for *n* features from $O(n!)$ to $O(n2^{n-1})$. Further, we discovered that encoding this state-space reduction technique into Promela programs was problematic: the programs became complicated, and ultimately did not reduce the likelihood of exhausting memory. It seemed easier for us to write our own tool (avoiding Spin altogether) to verify that models are contradiction free by retracing previous computations, to substantially reduce the memory requirements for model verification. This lead us to our second solution.

5. Incremental Consistency Algorithm

A feature model is *k-contradiction free* if every selection of *k* features does not expose a contradiction. A model of *n* features is *contradiction free* if it is *k*-contradiction free for all *k* where $0<k\leq n$. (Note that "unconditionally" dead features are exposed when $k=1$ [6]).

Suppose a sequence of features has been selected (and their consequences are propagated to the selection or deselection of other features). Figure 6 lists the `lookAhead` algorithm which determines if the selection of the next feature (for all such features) exposes a contradiction; it returns `true` if there is no contradiction, `false` otherwise. `BCP` denotes the boolean constraint propagation algorithm, which returns `true` if no contradiction was encountered in propagating constraints, `false` otherwise.

Let `V` denote the set of all features. A model is `k`-contradiction free if `lookAhead()` returns `true` for all subsets `S`⊆`V`, where `|S|=k-1`. That is, each subset `S` contains precisely `k-1` features that were selected. We generate sequences of length `k-1`, where each sequence corresponds to precisely one set of size `k-1`, and we guarantee that no two sequences of have the same feature membership. The challenge in enumerating sequences is to account for constraint propagation. For example, consider `k=2`. We do not consider the sequence `(A,B)` if selecting feature `A` automatically selects (via constraint propagation) `B`. The sequences we generate must be sequences that users of a feature-selection tool could produce. Figure 7 (on next page) sketches the algorithm we used to prove a model is *k*-contradiction free; it generates all sequences of length *k-1* observing the above constraint.

```
// let V be the set of variables (features) along with
// their current truth assignment (T,F,U).
// lookAhead returns true if the selection of the next
// feature does not expose a contradiction

boolean lookAhead() {
   Vreset = V;          // checkpoint (save)
                        // existing truth assignments
   foreach var in V {
      if (var==U) {     // if var (feature) not selected
         var=T;         // select it
         if (not BCP()) // propagate constraints
            return false;// contradiction was found
        V = Vreset;     // rollback (restore) assignments
      }
   }
   return true;         // no contradiction found
}
```

Figure 6 LookAhead Algorithm

```
// returns true if a feature model is k-contradict-free

boolean contradictionFree( int k ) {
   return contradictionFree(k,0);
}
// The index of a variable in V is its rank.  Ranks are
// used to compute sequences of k features, where a
// sequence is generated only once. contradictionFree(k,r)
// returns true if all possible selections of k features
// (whose rank is <=r) does not expose a contradiction

boolean contradictionFree(int k, int r) {
   if (k==1)                // we selected a set of features
      return lookAhead();   // lookahead for a contradiction
   else {
      // let g be the # of unselected variables
      // in V with rank>r
      if (g<k)
         return true; // no way to select k features when
                      // there are only g features remaining
     // otherwise select another feature
      Vreset = V;                   // checkpoint/save state
      foreach var in V {
         if (var==U and rank(var)>r) {
            // if feature is not yet selected and
            // its rank is past r, select it
            var=T;
            if (not BCP())
               return false;
            // no contraditions yet. select another feature
           if (not contradictionFree(k-1,rank(var)))
               return false;
            V=Vreset;               // rollback/restore state
         }
      }
   }
}
```

Figure 7 Contradiction Free Algorithm

Our incremental consistency algorithm (**ICA**) verifies that a model is contradiction free if it is *k*-contradiction free for all *k* where $0<k\leq n$ (Figure 8). *Note: when k=1, ICA exposes unconditionally dead features. When k>1, ICA exposes conditionally dead features.* To express the "coverage" of a search space, we print the value of each *k* for which *k*-contradiction freedom has been proven. When *k*=*n*, where *n* is the number of user selectable features, the model has been proven to be contradiction free.

An important optimization of our algorithms is saving and restoring the values of variables (i.e., saving and restoring the variable array **v** in the **lookAhead** and the **contradictionFree** algorithms). Instead of creating a stack where we pushed and popped the state of **v**, we simply remembered the set of variables that changed since the last "save state" or checkpoint. Generally few variables (features) change values upon selecting a feature and propagating its consequences. Undoing (restoring to a checkpoint) is fast.

Another observation is that we mistakenly thought the BCP algorithm consumed little CPU. Originally we implemented our algorithms without checkpoints. When we computed a sequence of features, we reset the **v** array to its initial state and recomputed the state of **v** by selecting each feature in order and propagating constraints.

The above two optimizations had a significant effect on the performance of ICA. Figure 9 shows the execution times of ICA for both the optimized and the unoptimized versions, along with the exhaustive Spin numbers. In almost all test cases, the *unoptimized* ICA generally executes noticably faster than Spin, *and* provides answers to conflict freedom in many cases where Spin failed to produce an answer. The optimizations that we described above (listed in the ICA Optimized column) provide the same solutions as the unoptimized ICA with an order of magnitude or more increase in speed. However, even with ICA optimizations, we were not able to determine whether two models (BerkeleyDB and Violet) were free of contradictions.

Figure 10 shows how far our optimized ICA algorithm was able to prove contradiction freedom. The BerkeleyDB

```
// let V be the set of all user-selectable
// variables/features and V.sizeof = n, the number of
// all user-selectable features. ICA returns true if
// the feature model is contradiction free

boolean ICA() {
   for k = 1 to V.sizeof {
      if (not contradictionFree(k))
         return false;
      print("Coverage up to "+k);
   }
   return true;
}
```

Figure 8 Incremental Consistency Algorithm

Model Name	Spin Exhaustive (secs)	ICA Unoptimized (secs)	ICA Optimized (secs)
BerkeleyDB	∞	∞	∞
Folutest	3.5	7.7	0.6
Freeman	1.2	0.01	0.01
GG4-model	8.6	13.6	1.8
GPL	∞	80	10.3
Notepad	∞	1916.5	118.5
SVGMapApp	∞	532.8	10.8
TightVNC	∞	2135.5	28.9
Violet	∞	∞	∞
apl	4.8	0.05	0.02
long	1.6	0.02	0.02

Figure 9 Execution Time of ICA for Different Models

k-contradiction free	BerkeleyDB (in hours)	Violet (in hours)
1	0.00	0.00
2	0.01	0.01
3	0.04	0.20
4	0.20	1.80
5	0.86	11.16
6	3.22	–
7	11.52	–

Figure 10 Coverage of ICA

model has 55 selectable features; in one hour of computation, we were able to prove that it was 5-conflict free. The Violet model has 64 selectable features, and in one hour we were able to prove it was 3-conflict free. In short, we hardly covered any of the state space of these models. We believe the ICA algorithms can prove conflict freedom in models that have about 20 (or fewer) features; ICA seems ineffective in models with substantially larger number of features.

6. Perspective and Related Work

We realized that the difficulty of statically finding contradictions in feature models can be explained from the following perspective. A standard analysis of feature models ensures that they are *satisfiable* or *non-void* — that the represented product line has at least one product [6]. (Stated differently, the propositional formula that encodes the feature model is satisfiable). Testing satisfiability is NP-complete. For a feature model to be contradiction free requires a *much* stronger property than satisfiability: each time a feature is selected and its consequences are propagated, the resulting predicate represents a simplified feature model, here called a *submodel*. (From the Introduction, model **M** is a submodel of **G** when **C** is selected). Contradiction freedom means that *every* possible submodel of a feature model is satisfiable or non-void (i.e., every submodel has at least one

product). Conversely, if a submodel is unsatisfiable (i.e., it has no products), then the original feature model has a contradiction (equivalently, a conditionally dead feature). For example, if a user is allowed to select feature **A** in model **M**, the resulting submodel is unsatisfiable. It is not clear if an efficient algorithm can be developed to statically identify model contradictions, due to the exponential number of submodels of a given feature model. In the interim, the engineering solution suggested in Section 2. can be used.

6.1. Related Work

Trinidad et. al [29] provide a framework for automating the error treatment of feature models, where feature models are expressed in terms of CSP (as opposed to propositional formulas as we have done). They focus on three types of errors. (1) A *dead feature* is a non-selectable feature, a feature that not appear in any product. (2) A child feature which is non-mandatory, is a full-mandatory feature if it is always chosen whenever its parent is chosen. And (3) a feature model is said to be *void* (which we called unsatisfiable earlier) if no product can be defined. The goal of [29] is to detect the above three errors and provide explanations for the relationships that caused these errors. Prior to our work, the relationship of dead features with void models (that is, a dead feature leads to a void submodel) was not previously known. Our work generalizes and unifies dead feature and void analyses to consider conditional errors.

[27][28][30] use BDDs to represent configuration models, generalizations of feature models that permit non-boolean attributes. The requirement that configuration tools be backtrack free (i.e., prevent users from uncovering model contradictions) is discussed in the context of using BDDs as a general engine for displaying user options at a particular state in a design, propagating constraints, and explaining inferences. We are unaware of work in the configuration modeling community that seeks analyses to find model contradictions [1][2][10].

Our notion of *k*-contradiction free was inspired by, but not identical to, the notion of *k*-consistency used by Kumar [19] and Freuder [13] for solving *constraint satisfaction problems (CSP)*. A model is *k*-strong consistent if, given a consistent sequence of *k-1* variables, any *k*th variable that is chosen from the set of unassigned variables has a value that satisfies all the constraints. While their goal is to find an overall consistent solution for a CSP, our goal is to find a particular sequence of user-selections that lead to a contradiction.

Marinov et.al [22], define boolean constraint propagation networks as an inference engine for implementing knowledge-based systems. They define inconsistencies in a model as a permanent conflicts between two chains of propositions (propagations) that always imply opposite values at a node (or a feature). They also stated that such problems should be detected at compile time, but no algorithms for doing so were presented.

Dynamic constraint satisfaction problems (DCSP) extends CSP to include evolving constraints because of assignments of values to variables. Soininen et. al [24] express configuration problems as DCSPs and also show that they are NP-complete. Verfaillie et. al [31] use dynamic backtracking to detect inconsistencies in DCSPs and also provide explanations in terms of the constraints that lead the system to an inconsistent state.

7. Conclusions

A feature model defines each product of a product line by a unique combination of features. A contradiction in a feature model is an error in a product line's specification. Finding such errors is important, both to model designers (as they have specified nonsensical constraints) and customers of the product line. We suggested a dynamic solution to find contradictions, where errors can be detected during usage and silently reported to model designers. However, the focus of this paper was to investigate whether model contradictions could be efficiently found by static analysis. We found that model checking could be used, but is so slow that only the simplest feature models could be verified. We then developed an *Incremental Consistency Algorithm (ICA)*, which incrementally verified increasingly stronger properties of contradiction freedom. Although our ICA was at least an order of magnitude faster than model checking, and that it verified contradiction freedom in more models, it too had practical limits. We believe that ICA can verify contradiction freedom of models with about 20 or fewer selectable features. In short, our experimental results suggest that a static analysis to find contradictions in feature models with large number of features may be very difficult.

Product lines are increasing common in software development. In the automotive industry, it is not uncommon for models to have hundreds, if not thousands of features [5]. Moreover, future feature models will not be limited to selecting (or even deselecting) individual features, but also supplying numerical constraints (e.g. performance bounds, cost bounds) on feature selections [5][6]. As feature models become more complex, the ability to guarantee that they are absent of certain kinds of errors becomes even more important. Finding contradictions in feature models is an interesting, practical, and basic problem. The contribution of our paper is a step toward more effective tools for feature model verification.

Acknowledgments. I gratefully acknowledge insightful conversations with D. Batory, D. Benavides, S. Krishnamurthi, and K. Fisler. I also thank the referees for their helpful comments. This work was supported by NSF's Science of Design Projects #CCF-0438786 and #CCF-0724979.

8. References

[1] AAAI Workshop on Configuration, 2003. `www2.ilog.com/ijcai-03/`

[2] AAAI Workshop on Configuration, 2007. `www.cs.ucc.ie/~osullb/aaai-config-ws-2007`.

[3] M. Antkiewicz and K. Czarnecki, "FeaturePlugIn: Feature Modeling Plug-In for Eclipse", *OOPSLA'04 Eclipse Technology eXchange (ETX)* Workshop, 2004.

[4] D. Batory. "Feature Models, Grammars, and Propositional Formulas", *SPLC 2005*.

[5] D. Batory, D. Benavides, and A. Ruiz-Cortes. "Automated Analyses of Feature Models: Challenges Ahead", *CACM*, Special Issue on Software Product Lines, December 2006.

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. "Automated Reasoning on Feature Models", *Conference on Advanced Information Systems Engineering (CAISE)*, 2005.

[7] D. Beuche, "Composition and Construction of Embedded Software Families", Ph.D. thesis, Otto-von-Guericke-Universitaet, Magdeburg, Germany, 2003.

[8] Big Lever, GEARS tool, `http://www.biglever.com/`

[9] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[10] ECAI 2006 Workshop on Configuration, `fmv.jku.at/ecai-config-ws-2006/`

[11] K.D. Forbus and J. de Kleer. *Building Problem Solvers*, MIT Press 1993.

[12] K.D. Forbus. email correspondence, 2007.

[13] E. U. Freuder. "Synthesizing Constraint Expressions". *CACM*, Vol 31, Issue 11, 1978.

[14] G.J. Holzmann. "The Model Checker Spin", *IEEE TSE* May 1997.

[15] G.J. Holzmann. "An Analysis of Bitstate Hashing", *Formal Methods in System Design*, Vol 13, 3, pp. 287-305, Kluwer, November 1998

[16] M. de Jong and J. Visser. "Grammars as Feature Diagrams", 2002.

[17] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Technical Report, CMU/SEI-90TR-21, November 1990.

[18] C. Kaestner, S. Apel, D. Batory. "A Case Study Implementing Features Using AspectJ", *SPLC 2007*.

[19] V. Kumar. "Algorithms for Constraint Satisfaction Problems: A Survey", *AI Magazine*, Vol 13, Issue 1, pp 32-44, 1992.

[20] R. E. Lopez-Herrejon and D. Batory. "A Standard Problem for Evaluating Product-Line Methodologies", *GCSE/GPCE 2001*.

[21] M. Mannion. "Using first-order logic for product line model validation". *SPLC 2002*.

[22] G. Marinov, V. Alexiev, Y. Djonev. "Boolean Constraint Propagation Networks", *Aritificial Intelligence: Methodology, Systems, and Applications (AIMSA'94)*, 1994.

[23] Pure-Systems. "Technical White Paper: Variant Management with pure::variants", `www.pure-systems.com`, 2003.

[24] T. Soininen, E. Gelle. "Dynamic Constraint Satisfaction in Configuration". In Configuration papers from the AAAI workshop, pp. 95-100. AAAI Technical Report WS-99-05.

[25] `www.spinroot.com`

[26] D. Streitferdt, M. Riebisch, and I. Philippow. "Details of Formalized Relations in Feature Models Using OCL". *ECBS 2003*, p. 297-304.

[27] S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Anderson, H. Hulgaard, J. Mffiler. "Comparing Two Implementations of Complete and Backtrack-Free Interactive Configurator", *Workshop on CSP Techniques with Immediate Application, International Conference on Principles and Practice of Constraint Programming. (2004) 97-111.*

[28] S. Subbarayan. "Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems", *Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2005.

[29] P. Trinidad, D. Benavides, A. Duran, A. Ruiz-Cortes, and M. Toro. "Automated Error Analysis for the Agilization of Feature Modelling", *Journal of Systems and Software* (in press).

[30] E. R. van der Meer, H. R. Andersen. "BDD-based Recursive and Conditional Modular Interactive Product Configuration". *Workshop on CSP Techniques with Immediate Application*, CP04, *International Conference on Principles and Practice of Constraint Programming*, 2004.

[31] G. Verfaillie, T. Schiex. "Dynamic Backtracking for Dynamic Constraint Satisfaction Problems", *ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications,* 1994.

Do SAT Solvers Make Good Configurators?

Mikoláš Janota
Lero, School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Abstract

A configuration process *is about finding a* configuration*, a setting, that satisfies requirements given by the user and constraints imposed by the domain. Feature models are used to record product domains and constraints imposed on individual products. As such constraints are in practice of complex nature, it is desirable to perform the configuration* interactively. *This article shows how to utilize a SAT solver in an interactive configuration process to provide support to the user.*

1. Introduction

Configurable complex systems have gained attention in recent years in such domains as the automotive industry and *feature models* are used to capture the intended product domain [8, 13].

Each feature corresponds to a distinctive aspect of a product and a customer specifies his requirements by defining the desired features. Features that were not specified by the customer, must be *configured* at the producer's side. When the number of features is large and dependencies between them are complex, this represents non-trivial tasks for the humans involved.

This is a clear motivation for *interactive* support during the configuration process — a tool provides hints or performs decisions throughout the whole process, alleviating the burden imposed on the human.

Other researchers targeted this problem before. Batory [1] uses the so-called *logic truth maintenance system*, whose disadvantage is that the approach does not guarantee *backtrack-freeness* — the user may hit a dead end during the configuration process.

Other approaches [12] use Binary Decision Diagrams (BDDs) [3] that guarantee backtrack-freeness but are known to suffer from space explosion as they are explicit representations of all possible configurations.

This article outlines a third path, building mainly on Batory's work, and that is it to apply a *SAT solver*, a tool for deciding the satisfiability of Boolean formulas (see e.g., [10, 6]).

The motivations to use a SAT solver is two-fold. Firstly, nowadays SAT solvers are extremely efficient despite the NP-completeness of the problem and still improving as illustrated by the yearly *SAT Competition*[1]. Secondly, a SAT solver is a step towards a more general *constraint solver* with the support for other than merely Boolean domains.

To the knowledge of the author, guaranteeing backtrack-freeness of and interactive configuration process by using a SAT solver is novel.

2. Background

Feature models [8] play an important role in *Software Product Line Engineering* [4]. In this article we are concerned only with feature models as descriptions of the product domain (also called *problem space*). In particular, we only need to know the semantics of the model that we are dealing with.

Various semantics of feature models exist, depending on their expressiveness [11, 7]. As we will be dealing with configuring a feature model, it is natural to use so-called *constraint satisfaction problem* (CSP) as the structure capturing the semantics; the following definition introduces the concept.

Definition 2.1 *A* constraint satisfaction problem *is a triple* $\langle X, D, \mathcal{C} \rangle$*, where* $X \equiv x_1, \dots, x_n$ *are variables,* $D \equiv D_1, \dots, D_n$ *are their respective domains and* $\mathcal{C} \subseteq D_1 \times \dots \times D_n$ *is a* constraint.

A valuation *is a function from variables to their domains, a valuation is called* partial *iff not all variables from* X *are assigned a value, it is called* complete *iff all variables from* X *are assigned a value.*

[1] http://www.satcompetition.org/

A solution v *to a constraint satisfaction problem is a complete valuation such that* $\langle v(x_1), \ldots, v(x_n)\rangle \in \mathcal{C}$.

The following toy example illustrates how feature models map to CSP, for more details on this topic see for instance the work of Benavides et al. [2].

Example 2.1 *A software application is developed for two markets: for Ireland and United States. The application deals with measurements and hence supports inches and centimeters. When released to the Irish market, centimeters must be used whereas inches must be used for United States. The corresponding CSP has two variables* $Country$ *and* $Units$ *whose domains are* $\{IE, USA\}$ *and* $\{cm, in\}$*, respectively. The constraint* $\mathcal{C}$ *comprises the following tuples (pairs in this case).*

$$\{\langle IE,\ cm\rangle, \langle USA,\ in\rangle\}$$

During the *configuration process* the user makes decisions by assigning values to variables and his goal is to find a solution to the given CSP.

In this article we will consider the scenario when a tool, let us call it a *configurator*, reacts upon every user's decision and provides some form of feedback. More specifically, we will be concerned with configurators that disable such values from variable domains whose selection would prevent finding a solution to the CSP.

In our small example, when the user selects Ireland as the desired country, the inches get disabled and centimeters selected automatically as that is the only option left.

The configurator is called *backtrack-free* iff it enables only the values for which there exists a solution, meaning that the user will never reach a dead end forcing him to retract some previous decisions. A configurator is called *complete* iff it does not disable values for which there exists a solution.

Formally, in each step of the configuration process, there is a user selection s_u — a valuation on variables $X_u \subseteq X$. The configurator is complete and backtrack-free iff for any $x_i \notin X_u$, the configurator disables the value v from the domain D_i if and only if there is no solution to the CSP such that it agrees with s_u on the variables X_u and assigns v to x_i.

2.1. Propositional World

A *propositional constraint satisfaction problem* is such CSP where each of the variable domains is $\{\text{TRUE}, \text{FALSE}\}$. This makes the job of the configurator somewhat simpler. If the configurator infers that a solution exists for only one of the two values, the variable must assume the second value and the user will not be allowed to change it; we will say that such variable is *locked*.

Note that if there is no solution for either of the two values for some variable, the given CSP does not have any solution since all the variables must be assigned to by a solution.

Rather unsurprisingly, any propositional constraint satisfaction problem $\langle X, D, C\rangle$ can be captured as a Boolean formula on the variables from X such that the valuations satisfying the formula correspond to the solutions of the problem.

Example 2.2 *Even though Example 2.1 is not given as propositional, it can be easily modeled as such. The following formula illustrates the principle.*

$$(IE \vee USA) \wedge (\neg\, IE \vee \neg\, USA) \wedge IE \Rightarrow cm \wedge$$
$$(in \vee cm) \wedge (\neg\, in \vee \neg\, cm) \wedge USA \Rightarrow in$$

As we are interested in the existence of solutions, it means that we are interested in satisfying a Boolean formula. To this end we will utilize a *SAT solver*. A SAT solver accepts as input a formula in so-called *conjunctive normal form* (CNF). A formula is in CNF iff it is a conjunct of disjunctions of variables or negated variables, e.g., $(\neg a \vee b) \wedge (\neg b \vee c)$. Each of the conjuncts, e.g., $\neg a \vee b$, is called a *clause* and the disjuncts are called *literals* (a single literal is also considered a clause). An important property of CNF is that evaluates to TRUE. If such valuation does not exist, hence the formula is unsatisfiable, it produces a proof of the unsatisfiability.

3. A SAT solver in Configuration Process

Our goal here will be to device an algorithm TEST-VARS that is called at the beginning of the configuration process and then after each user's decision.

We will approach the problem wielding the following armory. For any variable we can call the procedures LOCK and UNLOCK to disable and enable, respectively, the user to set the variable's value. Further, SET($variable$, $value$) sets a value for a variable; RESET($variable$) brings the variable to an unassigned state.

The SAT solver is represented by the function SAT(ψ) that returns either **null** iff ψ is unsatisfiable or it responds with a set of literals (negated or unnegated variables) that correspond to a satisfying valuation of ψ. For example, it may respond $\{a, \neg b\}$ to the query $a \vee b$.

The state of the configuration process and the constraint is captured by the formula ϕ. So for instance, if the CSP under concern is represented by the formula ψ and the user sets the variable v_1 to TRUE and the variable v_2 to FALSE, the formula ϕ will be equal to $\psi \wedge v_1 \wedge \neg v_2$.

Figure 1 presents the skeleton of the algorithm. For each variable that has not been set by the user, it computes whether there are satisfying valuations having the variable

```
TEST-VARS()
 1  foreach x that was not assigned to by the user
 2      do CanBeTrue ← TEST-SAT(φ, x)
 3         CanBeFalse ← TEST-SAT(φ, ¬x)
 4         if ¬(CanBeTrue ∧ CanBeFalse)
             then error "Unsatisfiable constraint!"
 5         if ¬CanBeTrue then SET(x, FALSE)
 6         if ¬CanBeFalse then SET(x, TRUE)
 7         if CanBeTrue ∧ CanBeFalse
 8           then RESET(x)
 9                UNLOCK(x)
10           else LOCK(x)
```

Figure 1. Basic version

```
TEST-SAT(φ: Formula, l: Literal) : Boolean
  if l ∈ KnownValues then return TRUE
  if l ∈ DisabledValues then return FALSE
  L ← SAT(φ ∧ l ∧ ⋀_{k∈DisabledValues} ¬k)
  if L ≠ null
    then KnownValues ← KnownValues ∪ L
    else DisabledValues ← DisabledValues ∪ {l}
  return L ≠ null
```

Figure 2. Improved version of TEST-SAT

set to TRUE and FALSE, respectively. The four possible combinations are investigated: If neither of them exists, then ϕ itself is unsatisfiable (line 4). If exactly one exists, then the variable is enforced to have the value for which there *is* a satisfying valuation (lines 5, 6) and the variable is locked (line 10). If both exist, the algorithm makes sure the variable is in the default state, i.e., unassigned and unlocked (lines 8, 9).

A straightforward implementation of TEST-SAT is simply to call the SAT solver on the conjunct of the given formula and literal: $\phi \wedge l$. This yields an algorithm which calls the solver twice on each variable, so it is warranted to investigate if that can be improved.

First let us look at the situation when the call to the SAT solver returns a satisfying valuation for $\phi \wedge l$. Consider a variable x_i which has been assigned the value TRUE by that valuation. We can conclude that $\phi \wedge x_i$ is satisfiable as $\phi \wedge l \wedge x_i$ is satisfiable. Therefore, there is no need to call the solver on $\phi \wedge x_i$. If x_i was assigned the value FALSE, analogous reasoning is carried out.

Hence, the first improvement to the algorithm is to store the values of which we already know that they appear in satisfying valuations encountered so far.

Can, on the other hand, the negative response of the solver be useful? Yes it can! Consider the situation when the constraint contains the formula $x_1 \Rightarrow x_2$. If the solver knows that x_2 cannot be TRUE, it can quickly deduce that x_1 cannot be TRUE either[2]. This motivates the second improvement: storing the values that were disabled and conjoing them to the overall formula in subsequent queries.

Figure 2 presents the procedure TEST-SAT. Literal l in inserted into the set $\mathit{KnownValues}$ if $\phi \wedge l$ if satisfiable. Analogously, literal l is added to the formula $\mathit{DisabledValues}$ if $\phi \wedge l$ is unsatisfiable.

When and how do we initialize the two sets? The safest (and coarsest) approach is to empty both sets whenever ϕ changes, i.e., at he beginning of each execution of TEST-VARS. However, if we know that ϕ has been strengthened, which happens for instance when a user sets a value of an unassigned variable, we can keep $\mathit{DisabledValues}$ and empty $\mathit{KnownValues}$. Analogously, if ϕ is weakened, e.g., when a user's decision is retracted, we can keep $\mathit{KnownValues}$ and empty $\mathit{DisabledValues}$. For further discussion on this topic see Section 6. To conclude, the following two procedures show how TEST-VARS is invoked.

```
ASSERT-DECISION(l : Literal)
  decisions ← decisions ∪ {l}
  φ ← ⋀_{l∈decisions} l
  KnownValues ← ∅
  TEST-VARS()
```

```
RETRACT-DECISION(l : Literal)
  decisions ← decisions ∖ {l}
  φ ← ⋀_{l∈decisions} l
  DisabledValues ← ∅
  TEST-VARS()
```

3.1. Producing Explanations

For each locked variable a configurator should explain to the user *why* it was locked. The explanation should contain the user's decisions that led to the lock, but even better, the relevant parts of the constraint.

In our case, when the constraint is in CNF, obvious candidates for parts of the constraint are the individual clauses (disjunctions of literals).

Recall that a value is locked if the solver returns unsatisfiability for the other value. For example, if a variable was locked in the TRUE value, then it must have been shown

[2] The solver detects that from the clause $\neg x_1 \vee x_2$ the literal $\neg x_1$ must be TRUE to satisfy the clause. This technique is known under the name *Unit Propagation* and is an essential part of state-of-the-art solvers.

that there is no solution with the variable having the value FALSE. The *proof* of this unsatisfiability is exactly the explanation we are looking for.

Obtaining a proof from a SAT solver is rather straightforward, see for example [15]. The proof is a subset of the input clauses whose conjunct is unsatisfiable.

However, a proof obtained from the solver might not be minimal in the sense that removing certain clauses or user-decisions will still yield an unsatisfiable formula. Naturally, for the user-friendliness sake, it is desirable for the explanation to be small.

To this end a fast technique with good results was proposed by Zhang and Malik [14], which calls the solver again on the proof that it has just produced. As the proof is an unsatisfiable formula, the solver will find it unsatisfiable and produce a new proof (possibly smaller than the original one). This process is iterated until the proof remains irreduced by the solver.

3.2 Example Execution

The following text illustrates the algorithm's workings on a constraint represented by the following formula, which we will denote as ψ.

$$\begin{array}{l} p \Rightarrow (q \Rightarrow \neg r) \wedge \\ p \Rightarrow (\neg q \vee \neg r) \end{array}$$

The procedure TEST-VARS begins by testing the variable p. Since $\psi \wedge p$ is satisfiable, the SAT solver finds a satisfying valuation $\{p, \neg q, \neg r\}$, let's say, whose literals are added to $KnownValues$. Similarly, the query $\psi \wedge \neg p$ yields the literals $\{\neg p, q, r\}$ which are also added to $KnownValues$.

Thanks to the luck we had in finding satisfying valuations, the tests for the variables r and q do not need to use the SAT solver as all four pertaining literals are already in $KnownValues$.

Now the user selects the feature p (assigns TRUE to p). Which yields a formula equivalent to the following one.

$$\begin{array}{l} (q \Rightarrow \neg r) \wedge \\ (\neg q \vee \neg r) \end{array}$$

Both values are valid for the variable r under this constraint. However, the SAT solver responds that q cannot be set to TRUE and hence the algorithm locks it in the value FALSE. Note that this is not detected if only unit propagation is performed as in [1].

4. Implementation

The author implemented the presented ideas and the implementation can be found our research group's website [3].

[3] http://kind.ucd.ie/products/opensource/Config/releases/

Figure 3 offers a screenshot of the application. A subtle difference from the presentation in Section 3. Due to the check-box user interface, that the user can only *select* features — set variables to TRUE— and *deselect* features — set TRUE variables to unassigned. Consequently, user decisions comprise merely non-negated literals.

Apart from automated selections and explanations, the application detects which constraints still remain to be satisfied. In the image we can see that the type of engine and gearshift are yet to be specified. Another functionality is the *configuration completion*, which fills in some values, determined by the SAT solver, that satisfy the unsatisfied constraints.

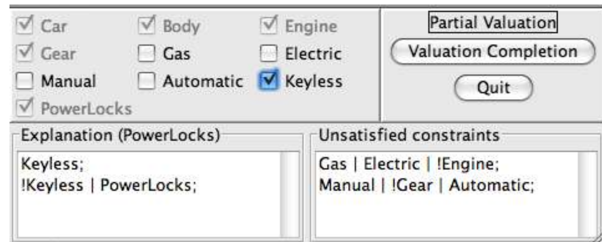


Figure 3. Configuration in action

The program is written entirely in Java and a SAT solver was written as well. Even though a third-party SAT solver could have been used, the author felt as he needs to write one to better understand what he's doing. Moreover, the proof generation is harder to access in other solvers as they need to account for enormous proofs and hence typically record them on the disk.

Nevertheless, it wouldn't be difficult to adapt an existing SAT solver for the same purpose; MINISAT, for instance, is well documented, open-source, and ranks well in competitions [6].

4.1 Experimental Results

To test the feasibility of the approach, the author has ran the prototype on an example feature model representing a car with 10 different features published in [5]. The average time of the algorithm TEST-VARS was 26ms and 41ms was the worst time.

Further, pseudo-random CNF formula was generated with 70 variables and 100 clauses where each clause has 3 literals. The average time was 107ms and the worst 290ms.

The tests were performed on a 1.67 Ghz PowerPC with 1 GB of RAM running Mac OS X.

As the SAT solver used is significantly slower than the top ones, these initial results are promising.

5. Summary and Discussion

The article presents how to use a SAT solver to implement an interactive and backtrack-free tool support for configuration process. Compared to using BDDs, this approach is *lazy*, so to say. A BDD is a pre-compiled representation of the configuration space and it is easy to find small Boolean formulas that explode the BDD in size. On the other hand, to heavily burden a modern SAT solver with a small formula is quite difficult.

Hence, the author believes that the SAT solver approach has a better chance of scaling.

A disadvantage of SAT solvers is that they require a specific input format — typically CNF. That can be overcome by *clausifying* the input, which can be done in such way that the output is linear w.r.t. input. However, such conversion would complicate the explanation generation. For BDDs, this is even worse as the formula's structure is lost.

6. Future Work

Efficiency The algorithm presented in Section 3 is not exchanging information between different calls for different user selections as much as it could have. It could be easily improved by looking at the proofs of locked variables. As long as the user does not alter the decisions on which a proof depends, the pertaining variable will remain locked.

Generalization It is not hard to see how the algorithm TEST-VARS could be generalized for variables with other than just two-valued domains. The algorithm would iterate over each domain, testing each value and eventually analyzing the cardinality of the reduced domain. It is clear, however, that for large domains this would be computationally infeasible and hence a more sophisticated technique is required for such. Most likely, calls to the solver would query for multiple values at a time.

7. Acknowledgments

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1.

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *SPLC '05*, LNCS. Springer-Verlag, 2005.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In P. Oscar and J. a. Falcão e Cunha, editors, *Proceedings of 17th International Conference on Advanced Information Systems Engineering (CAiSE 05)*, volume 3520 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley Publishing Company, 2002.

[5] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In Kellenberger [9].

[6] N. Eén and N. Sörensen. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT '03)*. Springer-Verlag, 2003. Available at http://www.een.se/niklas.

[7] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In Kellenberger [9].

[8] 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-021, SEI, Carnegie Mellon University, Nov. 1990.

[9] P. Kellenberger, editor. *Software Product Lines*. IEEE Computer Society, 2007.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC '01)*, 2001.

[11] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceeding of 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.

[12] S. Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proceedings of the CP-AI-OR*. Springer-Verlag, 2005.

[13] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.

[14] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Proceedings of SAT*, 2003.

[15] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2003.

On SAT Technologies for dependency management and beyond

Daniel Le Berre Anne Parrain*
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
{leberre,parrain}@cril.univ-artois.fr

Abstract

SAT solvers technology is now mature enough to be part of the engineer toolbox side by side with Mixed Integer Programming and Constraint Programming tools. As of June 2008, two great pieces of software are using SAT technology to manage dependency like problems: the open source Linux distribution OpenSuse 11.0, released on June 19, 2008, integrates a custom SAT solver in their dependency manager Zypp. The new update manager of Eclipse 3.4, called Equinox p2, also uses SAT technology to resolve dependencies in their OSGi platform. The use of SAT technology in Software Product Lines has already been pointed out by several authors. We believe that the current interest for solving dependency management problems in SAT technologies opens quite interesting challenges to the SAT community. First, since those problems are met in software with interactive use, SAT engines need to solve them within seconds. Second, providing one solution is usually not sufficient: finding the best solution is usually what users want. Finally, fully supported open source or commercial SAT engines are needed for a broader adoption in the software engineering community.

1 Introduction

The success of SAT technology in Electronic Design Automation (EDA), in particular in the area of Bounded Model Checking [6] has pushed forward the development of SAT solvers. They are now being used routinely as lightweight constraint programming solvers. The performance breakthrough due to the Chaff SAT solver[19], and the design some years later of the minimalistic Minisat [13] allowed the availability of numerous very efficient SAT engines in various languages. One of the main advantage of SAT solvers against more traditional constraint programming solvers is its simple unified input format (Dimacs [15]) that allows SAT solvers to be used just as black boxes, or as SAT components in a more software engineering vocabulary.

If many users of SAT solvers are currently coming from EDA, the use of SAT technology is currently growing in software engineering. Recent works in software design analysis clearly focus on efficient translation into SAT [23] and take advantages of new features developed in SAT solvers (e.g. Unsat cores [22]) to improve both user experience and scalability. In Software Product Lines, SAT solvers can be used to decide whether a product configuration is safe or not and how to implement the product configuration [3, 21]. Related work includes the use of CSP and BDD approaches to tackle the same problems [4, 5]. The EDOS project [16, 11] found in SAT technology a good mean to validate the package dependencies for Linux distributions, i.e to answer the question:"Are all the packages of that distribution installable?". In the same spirit, OPIUM [24] is a dependency manager for the Linspire linux distribution based on pseudo boolean solvers, one of the numerous extensions to SAT.

As a consequence of such research work on dependency management, two great pieces of software are using SAT technology to manage dependency like problems: the open source Linux distribution OpenSuse 11.0 [1], released on June 19, 2008, integrates a custom SAT solver in their dependency manager ZYpp. The new update manager of Eclipse 3.4, called Equinox p2, also uses SAT technology to resolve dependencies in its OSGi platform [2]. Finally, another famous framework for Java users, Maven, decided recently[3] to use SAT technologies to resolve their package dependencies.

*This work is supported in part by the European research project number 214898 Mancoosi (www.mancoosi.org).

[1] `http://en.opensuse.org/OpenSUSE_11.0`
[2] `http://wiki.eclipse.org/Equinox_P2_Resolution`
[3] `http://jira.codehaus.org/browse/MARTIFACT-20`

2 Dependency decision problem

Dependencies between packages can be easily modeled using propositional logic. For instance, package A depends on package B and package C can be expressed by the logical formula $A \rightarrow B \wedge C$ which in turn can be expressed by the two clauses $A \rightarrow B$ and $A \rightarrow C$ (or $\neg A \vee B$ and $\neg A \vee C$). If all the dependencies are requirements of a conjunctive form, even if incompatibilities between packages are expressed, the resulting SAT problem is made of Horn clauses, i.e. clauses containing at most one positive literal, thus is solvable in linear time [10].

The dependency problem becomes interesting when a given feature can be provided by several artifacts: this is the case for instance of several versions of the same OSGi bundle[4] in Eclipse. Sometimes, the same feature is provided by different packages, depending on their origin: to install latex in a linux distribution, one can use for instance texlive-latex or tetex-latex. In that case, we would express such dependency by something like $latex \rightarrow texlive_latex \vee tetex_latex$ which is no longer a Horn clause. Thus the dependency problem becomes NP-complete [8].

However, SAT solvers can nowadays solve some instances of the SAT problem with as many as millions of variables and clauses while they were still unable to solve a custom crafted 117 variables only SAT instance during the SAT 2007 Solver competition [5]. So, while there is no warranty that a SAT solver can solve efficiently SAT instances resulting from the translation of a "real" problem into SAT, it is often the case that such solvers perform well on those instances. From our own experience, the dependency decision problem can be easily solved using modern SAT solvers.

3 From satisfaction to optimization

However, the dependency problem is often *under-constrained*, which means that there is usually a lot of possible solutions. And not all those solutions are usually equally good. For Linux distributions for instance, one could take into account the number of packages to install, or their size, etc in order to propose an installation with the minimum number of packages or an installation with the smallest footprint on the hard drive. Those criteria can be easily modeled in an optimization framework by an objective function to minimize. Such objective function is of the form $\sum_{i=1}^{n} a_i x_i$ where a_i is an integral coefficient and x_i is a propositional variable where true is denoted by value 1 and false is denoted by value 0.

If one wants to minimize the number of installed packages, all a_i will be 1 and all x_i will correspond to the propositional variables encoding packages to install. If one wants to minimize the size of the installed packages, then the a_i will encode the size of each packages in bytes for instance.

In the case of SPL, the objective function could encode to install as many features as possible, to look for the cheapest or the most expensive product, etc.

Adding such objective function to a set of clauses creates an instance of the Binate Covering problem [9]. Such problem has already been studied in EDA for logic synthesis (to minimize the number of components needed to perform a given operation). From a complexity theory, that problem is NP-hard, which means that is is at least as hard as SAT.

The binate covering problem can be seen as a very specific case of an Integer Linear Program in which case efficient ILP frameworks exist (e.g. CPLEX). However, it is currently not clear if ILP solvers are the right approach to tackle those problems.

Indeed, ILP restricted to boolean variables has been also a recent area of research in the SAT community, under the generic name "Pseudo Boolean problems", inherited from the very first work on that subject [1, 2]. A competition of Pseudo Boolean solvers has been organized to assess the efficiency of existing solutions[18]. Among the current best ones, one can note the pure SAT approach of Minisat+[12], the hybrid approach Pueblo [20] (used in the tool OPIUM) or the modern branch-and-bound Bsolo [17]. Those solvers have been compared to the Gnu Linear Programming toolkit, and in many cases performed better. No comparison with commercial ILP solvers has been done yet.

In Artificial Intelligence, the binate covering problem is sometimes presented as a so called "Weighted partial MAX-SAT problem": the clauses of the original binate covering problem are called hard clauses, i.e. they must be satisfied. The objective function is encoded by weighted unit clauses, whose weight is a_i and whose literal is $\neg x_i$. If the weight is the same for all the soft clauses, the problem is called "partial MAXSAT". Since 2006, there is an annual competition of MAXSAT solvers. The interest on MAXSAT solvers is currently growing so MAXSAT solvers will be yet another way to tackle the binate covering problem in the future.

SAT solvers are currently efficient enough to solve SAT

[4] OSGi is the component based model component chosen by Eclipse for its plug-in architecture. See `http://www.osgi.org/` for details.

[5] `http://www.satcompetition.org/2007/`

instances mapping real useful problems. Regarding solvers for SAT extensions like Pseudo Boolean or MAXSAT, the picture is less clear. Furthermore, we have seen that the so-called binate covering problem can be solved in many-ways: the best solution is yet to be determined.

The evaluation of the solvers is currently done on the assumption that the solvers are used in batch mode, i.e. that they can take several minutes, if not hours, to answer. Indeed, it is often the case in the area of model checking that the engineer writes its model during the day and let the SAT solver to check it during the night. In the SAT competition for instance, the timeout used in the industrial category is set to 1200 seconds in the first stage, and 10000 seconds in the second stage. As a consequence, the use of SAT solvers in interactive tools such as Eclipse or Linux update managers, or a Feature Model Editor such as FeatureIDE or FAMA requires specifically tailored solvers (see e.g. custom SAT engines in EDOS (debian) or OpenSuse.

Finally, the notion of quality of the solution is also a hot topic in that case: it is unlikely that a solver can efficiently solve all those NP-hard problems within a second, so non optimal solutions need to be returned after that time. We enter here in the area of anytime algorithms, for which local search algorithms are usually pretty good [14].

4 When modeling matters

The expected results mentioned in the previous section were simple, and global. However, in real applications, the user preferences are usually manifold and expressed in a local manner. Take for instance the case of the use of SAT technology in Eclipse p2: one expected result of the objective function is to make sure that most recent versions are installed preferably to older ones. If there is only one package, this is not a problem. However, as soon as there is more than one package, we enter the area of multi-objective/criteria optimization.

Indeed, suppose that package A is available in three versions: A_1, A_2, A_3. Suppose that package B is available in two versions: B_1, B_2. Suppose that package X depends on A and B. The best solution would be to pick the latest versions, B_2 and A_3. However, they are incompatible. Is it better to take A_2 and B_2 or B_1 and A_3? There is maybe no way to answer that question. In that case, an automated solution to solve that dependency problem will answer one of them, without any assumption that can be made on the solution. If one can add a preference between A and B, then we can discriminate between the two solutions and favor the first solution for instance if B is more important than A or the second one if A is more important than B.

Artificial Intelligence and Operational Research have solutions to deal with such kind of problems. In the specific case of Eclipse p2, the preferences on the different plugin versions could be modeled for instance using the Qualitative Choice Logic [7].

5 Research toys or real tools?

There are numerous SAT/Pseudo Boolean/MAXSAT solvers out there, in various languages, with various features. However, very few are really supported in the sense that there is no clear way to get help, enter bug reports, etc. And very few are really open source: many solvers have a license restricting their use for research purpose only. Most of them are not full featured in the sense that if they all allow to solve the SAT decision problem, very few support also proof logging, minimal unsat core, model enumeration, model counting, optimization support, support for multi-core processors, etc. In that sense, users of SAT technology must first define all the features they really need in order to choose the right solver (or the right solvers, since it might be just impossible to find a solver with all needed features). In that sense, SAT solvers are still research toys.

On the other hand, various products (from both academia and private companies) are now based on currently available SAT solvers. So the same solvers can be considered as real tools too.

The main current issue is certainly the input format used in the SAT community: the Dimacs format created for the Second Dimacs Challenge[15]. It is a simple integer based input format very convenient for SAT solver developers. It's simplicity is one key element of the incredible evolution of SAT solvers: everybody can easily read or produce SAT instances using that format, in any language. As a consequence, SAT solvers became quickly "black-boxes" fed using Dimacs formatted SAT instances. It was thus easy to compare the behavior of several solvers on the same benchmarks, organize sat competitions, etc. However, it is not a nice input format for people willing to try SAT technology: one needs to design an abstraction layer between its problem and the Dimacs format or the SAT solver first. The audience of SAT technology growing, a more user friendly input format is now necessary.

6 Conclusion

We have seen that SAT technology receives currently a lot of attention in the software engineering community, especially in the area of dependency management, that is also

a concern inherent to Software Product lines. We have seen that the basic decision problem related to dependency management is in practice easily solvable with current state-of-the-art SAT solvers. However, dependency management in real software is likely to be better modelled as an optimization problem called binate covering problem for which numerous solutions exist. We currently do not know the best approach for solving binate covering instances encoding the dependency problem. Another issue with dependency management lies in the fact that the problem is likely to be under-constrained, i.e. it is likely that the problem admit several equivalent solutions for the solver. As a consequence, it will be very difficult to control the answers provided by the solver, and to offer some warranties about those solutions. It must be clear for users of such technology that the most important part of the work is to properly express their problem in terms of preferences among possible solutions, then to deduce from those preferences the objective function of their binate covering problem. Finally, the use of solvers in interactive tools is likely to change the solvers landscape since most of them are currently designed to be used in batch mode.

References

[1] P. Barth. Linear 0-1 inequalities and extended clauses. Technical Report MPI-I-94-216, Max-Plank-Institut fr Informatik, Saarbrücken, Germany, 1994.

[2] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Max-Planck Institut fr Informatik, Saarbrücken, 1995.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[4] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. 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] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In R. Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[7] G. Brewka, S. Benferhat, and D. L. Berre. Qualitative choice logic. *Artif. Intell.*, 157(1-2):203–237, 2004.

[8] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[9] O. Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.

[10] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[11] The edos project. http://www.edos-project.org.

[12] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:1–26, 2006.

[13] N. E. en and N. Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[14] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[15] D. Johnson and M. Trick, editors. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[16] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.

[17] V. Manquinho and J. Marques-Silva. On using cutting planes in pseudo-boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:209–219, 2006. Research Note.

[18] V. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (pb'05). *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:103–143, 2006.

[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[20] H. M. Sheini and K. A. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:165–182, 2006.

[21] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[22] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[23] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[24] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

Automated Analysis of Feature Models using Atomic Sets *

Sergio Segura
Department of Computer Languages and Systems
University of Seville
Av Reina Mercedes S/N, 41012 Seville, Spain
sergiosegura AT us.es

Abstract

Scalability is recognized as a key challenge in the automated analysis of Feature Models (FMs). Current solutions in this context mainly propose using different logic paradigms as a way to improve the performance at the solution level while the problem remains the same. Atomic Sets (ASs) were proposed as a promising solution for the simplification of FMs (i.e. reduction of the number of variables) in the context of automated analysis. However, years after their introduction, the lack of specific algorithms and performance results still hinder its integration into current proposals and tools. In this paper, we set the basis for the usage of ASs as a generic technique for the automated analysis of FMs. In particular, we first propose a specific algorithm to construct the ASs of an FM. Then, we present a performance test measuring the degree of improvement (in time and memory) when implementing ASs into CSP, BDD and SAT-based solutions.

1. Introduction

The automated analysis of FMs enables the extraction of information from the models through the implementation of a number of analysis operations. The problems of feature combinatorics related to these operations are accepted to be NP-hard and can take a long time to solve [2]. Current solutions in this context mainly focus on the usage of different logic paradigms and solvers as a way to improve the performance at the solution level [2, 5, 6]. However, few efforts have been made to study how improving the performance through the treatment of FMs in the domain of the problem.

In [10], Zhang *et al.* proposed a propositional logic-based method for the verification of FMs at different binding times. In their work, the authors proposed the usage of so-called *atomic sets* as a way to improve the efficiency of the process at the problem level. An atomic set represents a group of features (at least one) that can be treated as a unit during the analysis of an FM. According to Zhang *et al.* working with atomic sets instead of features may often reduce the size of the problem to solve improving efficiency. However, even when atomic sets seems a promising technique, the benefits of using it are still unknown by the research community and the lack of specific algorithms and empirical results difficults its integration into current proposals.

In this paper, we set the basis for the usage of atomic sets as a generic technique for the automated analysis of FMs. To this aim, we report the lessons learned from integrating atomic sets into our framework for the automated analysis of FMs, FAMA[1] [5, 9]. In particular, we first propose an algorithm to construct the atomic sets of a given FM. Then, we detail the results of a performance test measuring the degree of improvement (in time and memory) when implementing atomic sets into CSP, BDD and SAT-based solutions integrated in FAMA.

The remainder of the paper is structured as follows: in Section 2, the automated analysis of FMs and atomic sets are introduced. Our algorithm for the computation of the atomic set of an FM and the experimental results are presented in Section 3. Finally, we summary our conclusions in Section 4.

2. Preliminaries

2.1. Automated Analysis of Feature Models

The analysis of an FM consists on the observation of its properties. Typical operations of analysis allow finding

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)

[1] http://www.isa.us.es/fama/

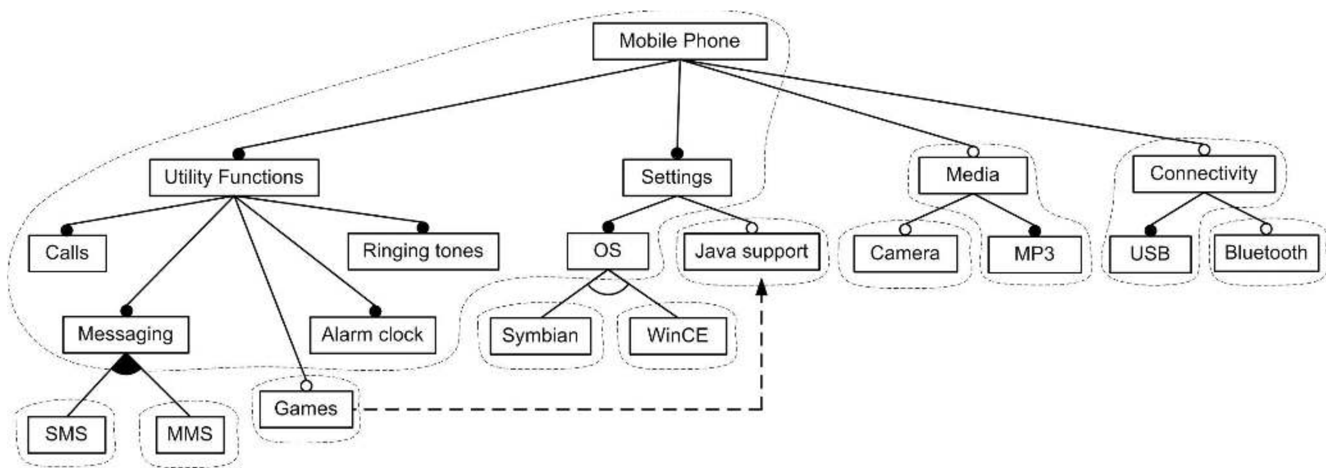


Figure 1. Atomic sets

out whether an FM is void (i.e. it represents no products), whether it contains errors (e.g. feature that can not be part of any products) or what is the number of product of the software product line represented by the model.

In order to enable the automated analysis of FMs specific logic paradigms and solvers have been proposed. In particular, the analysis is generally performed in two steps: i) First, the model is translated into an specific logic representation such as a Constraint Satisfaction Problem (CSP) [3], a SATisfiability problem (SAT) [1] or a Binary Decision Diagram (BDD) [8], ii) Then, off-the-shelf solvers are used to automatically perform a variety of operations on the logic representation of the model.

The available empirical results [5, 6] and surveys [4] suggests that there is neither an optimum logic paradigm nor solver to perform all the operations identified on FMs. As a result of this, current research efforts as the FAMA framework integrate different paradigms and solver in order to combine the best of all of them in terms of performance [7].

2.2. Atomic Sets

The usage of *atomic sets* for the automated analysis of FMs was proposed by Zhang *et al.* back in 2004 [10]. An atomic set represent a group of features (at least one) that can be treated as a unit during the analysis of an FM. The intuitive idea of atomic sets is that mandatory features and their parent features can be treated as a whole in certain analysis operation without altering the result. This is because those features can never appear in a product separately. Figure 1 depicts an FM inspired by the mobile phone industry. The areas delimited by dashed lines illustrate the atomic sets. As an example, feature *'USB'* can only be part of a product if its parent feature, *'Connectivity'*, is also part of the product. Thus, both features can be treated as a unit to perform some operation such as finding out if the model is void (i.e. it represents at least one product) or what is the number of product represented by the model. The step to use atomic sets as the basic unit in the analysis can be achieved by replacing each feature in the model by the atomic set which contains it.

The benefit of using atomic sets lies in efficiency. Working with atomic sets instead of features may reduce the number of variables to consider and consequently the size of the problem to solve. As an example, in the FM of Figure 1 the number of variables after constructing atomic sets is reduced from 20 (features) to 11 (atomic sets).

3. Our Contribution

3.1. Atomic Sets Computation

In this section, we present an algorithm for the construction of the atomic sets of a given FM. Figure 2 depicts the pseudocode of the function implementing our algorithm. The function receives an FM as input and returns a set of atomic sets as output. Each atomic set is composed by a name (e.g. "AS-2") and the collection of features in the atomic set. The main part of the work is made by a recursive procedure also presented as part of the figure. This procedure checks all the subfeatures of the feature received as input. For each subfeature, if it is mandatory, it is added to the current atomic set under construction. Otherwise, a new atomic set including the subfeature is created and added to the collection of atomic sets. After repeating this process with all the features in the model, the set of atomic set is returned.

```
1  function buildAS(FeatureModel fm)::Collection<AtomicSet>
2     Collection<AtomicSet> atomic_sets = new Set<AtomicSet>();
3     Feature root = fm.getRoot();
4     AtomicSet as = new AtomicSet("AS-0");
5     as.addFeature(root);
6     atomic_sets.add(as);
7     computeAS(atomic_sets, root, as, 0);
8     return atomic_sets;
9  endfunction

10 procedure computeAS(Collection<AtomicSet> atomic_sets, Feature f, AtomicSet current_set, int set )
11    foreach Feature g in f.getSubfeatures()
12       if (g.getRelationType() == Feature.MANDATORY)
13          current_set.addFeature(g);
14          computeAS(atomic_sets,g,current_set, set);
15       else
16          String setname = "AS-" + (set+1);
17          AtomicSet new_as = new AtomicSet(setname);
18          new_as.addFeature(g);
19          atomic_sets.add(new_as);
20          computeAS(atomic_sets,g,new_as,set+1);
21       endif
22    endforeach
23 endprocedure
```

Figure 2. Algorithm for the computation of atomic sets

3.2. Experimental Results

In order to measure the benefits of implementing atomic sets we carried out a performance comparison using the FAMA framework. In particular, we first generated a set of random FMs with a different number of features and cross-tree constraints. Then, we performed some operations using CSP, BDD and SAT-based solvers with and without the usage of atomic sets. Next, we gave the details about the experiment and the results.

3.2.1 The Experiments

For the experiments, we generated a number of random FMs using FAMA. The algorithm for the automated generation of those models is presented in Appendix A as one of the contributions of the paper. In particular, we used four groups of 50 randomly generated FMs. Each group included FMs with a number of features in a specific range ([50-100),[100-150),[150-200) and [200-300)) in order to test the performance with different sizes of the problem. Once all the FMs were generated, we proceeded with the execution using the FAMA framework. For the execution, we used three of the solvers integrated in FAMA: JaCoP[2] (CSP), JavaBDD[3] (BDD) and Sat4j[4] (SAT). The set of experiments was executed twice with each solver, with and without the usage of atomic sets, in order to compare the improvement in the performance. Each FM was executed several times increasing the number of cross–tree constraints from 0 to 5, 10, 15, 20 and 25% of the number of the features in the FM. Cross-tree constraints were added randomly as well, but checking that the same feature can not appear in more than one cross–tree constraint and that a feature can not have a cross–tree constraint with any of its ancestors. Once the results were obtained, we worked out averages from the results in order to avoid as much exogenous interferences as possible. Averages were obtained from all the FMs in each range with the same percentage of cross–tree constraints. Table 1 summarizes the characteristics of the experiments.

N. of Features	N. of instances	CT constraints
[50-100)	50	[0%-25%]
[100-150)	50	[0%-25%]
[150-200)	50	[0%-25%]
[200-300]	50	[0%-25%]

Table 1. Experiments

For our experiments we performed two operations: $i)$ finding out if an FM is valid, that is, if it represents at last one product and $ii)$ finding out the total number of products represented by a given FM. The first one is one of simplest

[2] http://jacop.cs.lth.se/
[3] http://javabdd.sourceforge.net/
[4] http://www.sat4j.org/

operation while the second is the hardest one in terms of performance because it is necessary to work out the total number of possible combinations. The data extracted from the tests were:

- Average memory used by the logic representation of the FM (measured in Kilobytes).
- Average execution time to find one product (measured in milliseconds).
- Average execution time to obtain the number of products (measured in milliseconds).
- Time to compute the atomic sets (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.6.0_04. We ran our tests on a WINDOWS VISTA BUSINESS EDITION machine equipped with a 2.4Ghz Intel Core 2 microprocessor and 2048 MB of RAM memory.

3.2.2 The Results

The experimental results revealed a noticeable improvement in the performance of solvers when using atomic sets. The first evidence was a reduction in the average memory usage of 4% in JaCoP, 15% in Sat4j and 19% in JavaBDD. The operation to find one product was performed on average between 10% (JaCoP and Sat4j) and 20% (JavaBDD) faster using atomic sets. Similarly, the number of products of the FMs was computed on average between 5% and 20% faster using this technique.

The improvement in the performance was better observable with large FMs. As an example, Figure 3 illustrates the average memory and time used by JavaBDD in the range of 200-300 features. Improvements were especially considerable in the experiments with a higher number of cross-tree constraints. In these cases, experiments using atomic sets revealed an improvement of up to 25 seconds (28% of improvement) on average when finding the number of solutions.

The benefits of using atomic sets were especially noticeable when we studied the hardest cases in terms of performance. An example of this situation is illustrated in Figure 4. This figure presents the time and memory consumed by JavaBDD in the worst cases of the range of experiments between 200 and 300 features. In these cases, we found an improvement of up to 119 MB (in a total of 476) in memory and 7 minutes (in a total of 27) in the time to find the number of solutions.

Finally, we found that the time to compute the atomic sets of FMs was insignificant (0-10 ms) in all the cases.

3.2.3 Discussion

The improvement in time when adopting atomic sets was not easy to measure in part of the experiments. This was because many of these were not complex enough and the time to perform the operations was very low. This is the reason because we provided range of improvements for the time and we do not give accurate data.

The available empirical results shows that the memory usage of BDD solvers can be huge [5]. In fact, it seems to increases exponentially with the number of cross-tree constraints. In this context, our experimental results suggest that atomic sets could be used as a suitable strategy to ease the effects of this trend.

Our performance test showed that the usage of atomic-set may provide a great improvement of the performance specially when dealing with large FMs. However, we remark that the cost of implementing atomic sets is practically insignificant and can bring very positive results even in small specific cases (especially when these are hard to compute).

Finally, we remark that in this paper we focus on the performance provided by different solvers when implementing atomic sets. For an empirical comparison of the performance provided by the solvers presented in this paper we refer the reader to [5].

4. Conclusions

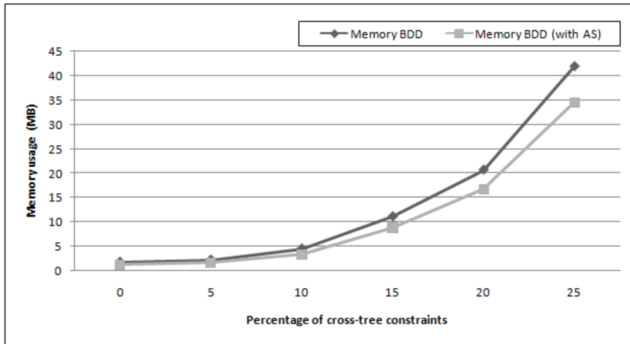
In this paper, we set the basis for the usage of atomic sets as a generic technique for the automated analysis of FMs. In particular, we first presented an algorithm to construct the atomic set of a given FMs. Then, we detailed the results of a performance test measuring the degree of improvement when using atomic sets with CSP, BDD, and SAT-based solutions currently integrated in the FAMA framework.

The experimental results revealed that the improvement when using atomic sets can be considerable in both, time and memory. This improvement was especially important in large FMs (in the order of hundreds of MB and minutes). However, we remark that the payoff for implementing this technique is insignificant and can bring noticeable results even in small cases, especially if these are hard to compute.

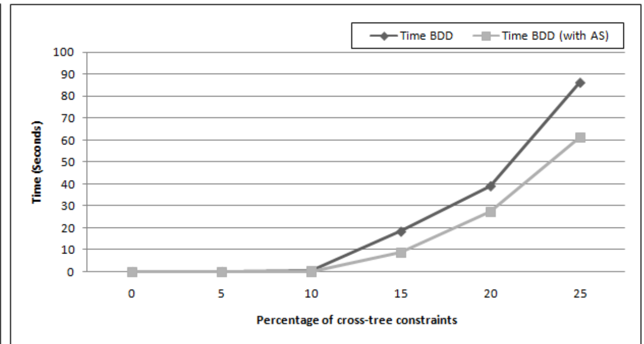
As a result of our performance test, we conclude that the techniques for the treatment of FMs at the problem level could help to improve the efficiency notably. Additionally, we consider that this kind of techniques could be applicable to the analysis of other kind of variability models.

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Confer-*

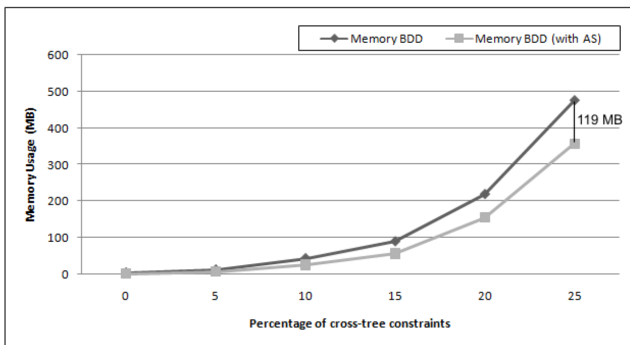


(a) Average memory usage

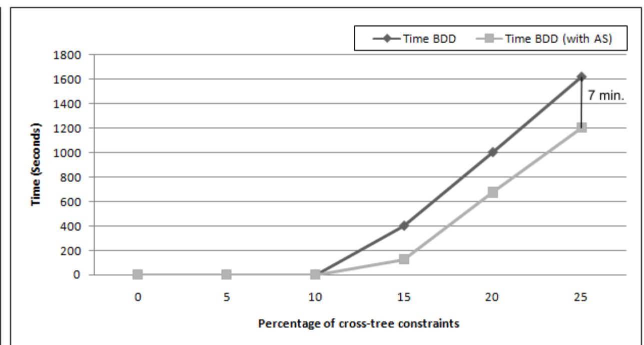


(b) Average time to find the number of products

Figure 3. Average time and memory usage of JavaBDD (range 200-300 features)



(a) Memory usage



(b) Time to find the number of products

Figure 4. Time and memory usage of JavaBDD in the worst cases (range 200-300 features)

ence, LNCS 3714, pages 7–20, 2005.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[4] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[5] 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.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[8] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005.

[9] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Proceedings of the 12th International Software Product Line Conference (Tool demonstration)*, 2008.

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

A Random Feature Model Generation

The random generation of feature models is one of the capabilities of the FAMA framework. Figure 5 depicts the pseudocode of the algorithm used for the generation of those models. In particular, the algorithm takes the following parameters as inputs:

w : is the maximum number of child relationships of the features in the model.

h : is the maximum height of the model. We consider the height of a feature model as the maximum distance between the root and any feature without considering cross–tree constraints.

e : is the maximum number of elements in a set relationship.

d : is the number of cross–tree constraints to be generated.

```
1   function GenerateFM(int w,h,e,d)::FeatureModel
2     FeatureModel fm = new FeatureModel();
3     Feature root = new Feature();
4     fm.setRoot(root);
5     fm.generateTree(w,h,e,root);
6     fm.generateCrossTreeConstraints(d);
7     return fm;
8   endfunction

9   procedure GenerateTree(int w,h,e, Feature parent)
10    if (h ≥ 1)
11      int nChildren = Random.getInt(w);
12      forach int i in {0..nChildren}
13        RelationType type = Random.getRelationType();
14        Feature child = new Feature();
15        switch(type)
16          case mandatory:
17            createMandatory(parent,child);
18            generateTree(w,h-1,e,child);
19          case optinal:
20            createOptional(parent,child);
21            generateTree(w,h-1,e,child);
22          case cardinality:
23            int card = Random.getInt();
24            createCardinality(parent,child,1,card);
25            generateTree(w,h-1,e,child);
26          case set:
27            int nChildrenSet = Random.getInt(e);
28            int setCard = Random.getInt(nChildrenSet);
29            FeatureGroup group = new FeatureGroup(nChildrenSet);
30            createSet(parent,group,1,setCard);
31          foreach Feature node in group
32            GenerateTree(w,h-1,e,node)
33          endforeach
34        endswitch
35      endforeach
36    endif
37  endprocedure

38  procedure generateCrossTreeConstraints(int d)
39    if (h ≥ 1)
40      int i = 0;
41      while (i < d)
42        Feature f = getFeatureRandomly();
43        Feature g = getFeatureRandomly();
44        if (valid(f,g))
45          if (Random.getBool == true)
46            generateDepends(f,g);
47          else
48            generateExcludes(f,g);
49          endif
50          i++;
51        endif
52      endwhile
53    endif
54  endprocedure
```

Figure 5. Algorithm for the generation of random feature models

Filtered Cartesian Flattening: An Approximation Technique for Optimally Selecting Features while Adhering to Resource Constraints

J. White, B. Doughtery, and D. C. Schmidt
Vanderbilt University, EECS Department
Nashville, TN, USA
Email:{jules, doughtery, schmidt}@dre.vanderbilt.edu

July 25, 2008

Abstract

Software Product-lines (SPLs) use modular software components that can be reconfigured into different variants for different requirements sets. Feature modeling is a common method for capturing the configuration rules for an SPL architecture. A key challenge for developers is determining how to optimally select a set of features while simultaneously honoring resource constraints. For example, optimally selecting a set of features that fit the development budget is an NP-hard problem. Although resource consumption problems have been extensively studied in the Artificial Intelligence and Distributed Computing communities, the algorithms that these communities have developed, such as Multiple-choice Knapsack Problems (MMKP) approximation algorithms, have not been extensively applied to feature selection subject to resource constraints. The paper provides the following contributions to the study of automated feature selection for SPL variants: (1) we present a polynomial time approximation technique called *Filtered Cartesian Flattening* (FCF) for deriving approximately optimal solutions to feature selection problems with resource constraints by transforming the problems into equivalent MMKP problems and using MMKP approximation algorithms, (2) we show that FCF can operate on large feature models that would not be possible with existing algorithmic approaches, and (3) we present empirical results from initial experiments performed using FCF. Our results show that FCF is 93%+ optimal on feature models with 5,000 features.

1 Introduction

Software Product-Lines (SPLs) [5] define reusable software architectures where applications are assembled from modular components. Each time a new version of an SPL application (called a *variant*) is created, the rules governing the composition of the SPLs modular components must be strictly adhered to. *Feature Modeling* [7] is a common modeling technique used to capture an SPL's configuration rules.

A feature model is a tree-like structure where each node in the tree represents a variation or increment in application functionality. Parent-child relationships in the tree indicate the refinement of application functionality. For example, in a feature model for an Magnetic Resonance Imaging system a parent feature would be *Magnet Strength* and the children would be *1.5 Tesla Magnet* or *3 Tesla Magnet*.

Each SPL variant is described as a selection or list of features that are present in the variant. The construction of a variant is bounded by adding constraints on how features are selected. If a feature is

selected, the feature's parent must also be selected. Moreover, features can be required, optional, or involved in XOR/Cardinality relationships with other features to model a wide range of configuration rules.

A key challenge when selecting features for an SPL variant is determining how to select an optimal set of features while simultaneously adhering to one or more resource constraints. For example, in a medical imaging system, if each hardware feature has a cost and value associated with it, selecting a set of features that maximizes the resulting variant's value but also fits within a customer's procurement budget is hard. Proofs that this problem is an NP-hard problem can be built by showing that any instance of the NP-complete *Multi-dimensional Multiple-choice Knapsack Problem* [1] (MMKP) can be reduced to a feature selection problem with resource constraints, as described in Section 3.1.

Existing techniques for deriving solutions to feature selection problems [4, 8, 3, 13] have utilized exact methods, such as integer programming [12] or SAT solvers [9]. Although these approaches provide guaranteed optimal answers they have exponential algorithmic complexity. As a result, these algorithms do not scale well to large feature selection problems with resource constraints.

Resource constraints have been extensively studied by Artificial Intelligence and Distributed Computing researchers. Although good approximation algorithms have been developed for problems involving resource constraints [1], these algorithms have so far not been applied to feature modeling. A key obstacle to applying existing MMKP algorithms to feature selection subject to resource constraints is that the hierarchical structure of a feature model does not fit into the MMKP problem paradigm.

To address the lack of approximation algorithms for selecting featues subject to resource constriants, we have created a polynomial-time approximation technique, called *Filtered Cartesian Flattening* (FCF), for selecting approximately optimal feature sets while adhering to multi-dimensional resource constraints. This paper provides several contributions to the automated construction of SPL variants. First, we present the polynomial time FCF technique for selecting approximately optimal feature sets while respecting resource constraints. We then show how FCF can be combined with different MMKP approximation algorithms to provide different levels of optimality and time complexity. Finally, we present initial empirical data showing that FCF provides roughly 93%+ optimality on feature models with 5,000 features.

The remainder of this paper is organized as follows: Section 2 describes the challenge of optimally selecting a set of features subject to a set of resource constraints; Section 3 presents our FCF approximation technique for selecting nearly-optimal feature sets subject to resource constraints; Section 4 presents empirical results showing that our algorithm averages 93%+ optimality on feature models of 5,000 features; Section 5 compares our work to related research; and Section 6 presents concluding remarks and lessons learned.

2 Challenge: Optimally Selecting Features Subject to Resource Constraints

A common goal in selecting features is to maximize the perceived value or quality of the variant produced. In the context of medical imaging systems, for instance, a key goal is to maximize the accuracy of the images produced. The problem is that there are usually additional constraints on the feature selection that are not captured in the feature model and that make the optimization process hard.

For example, features often have costs associated with them, such as the cost of different strength magnets for a Magnetic Resonance Imaging (MRI) machine. A producer of an MRI machine cannot force its customers to buy an MRI variant that exceeds the customer's budget. When a customer requests an MRI machine, therefore, the producer must select a set of features for which the sum is less than the customer's budget, which also maximizes the accuracy of the machine.

We call this problem *optimal feature selection subject to resource constraints*. As shown in Section 3.1, any instance of the NP-complete problem

(the MMKP problem) can be reduced to an instance of this problem. Optimal feature selection subject to resource constraints is thus NP-hard.

Large-scale industrial feature models, such as those for the automation software in continuous casting steel plants, can contain on the order of 30,000 features [11]. Existing techniques [4, 8, 3, 13] that use exact but exponential algorithms do not scale well to these large problem sizes. Other existing approximation algorithms, such as those for MMKP problems cannot be directly applied to optimal feature selection subject to resource constraints. Since exact algorithms do not scale well for these problems and existing approximation algorithms cannot be applied, it is hard to automate feature models for large-scale applications.

3 Filtered Cartesian Flattening

This section presents the *Filtered Cartesian Flattening* (FCF) approximation technique for optimal feature selection subject to resource constraints. FCF transforms an optimal feature selection problem with resource constraints into an equivalent MMKP problem and then solves it using an MMKP approximation algorithm.

3.1 MMKP Problems

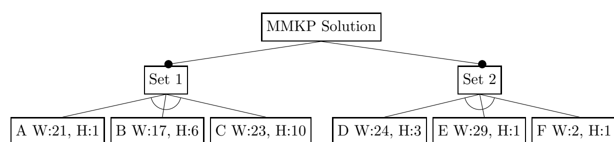


Figure 1: A Feature Model of an MMKP Problem Instance

A Knapsack problem [6] is an NP-complete problem where there is a knapsack of fixed size and the goal is to place as many items as possible from a set into the knapsack. An MMKP problem is a variation on the Knapsack problem where the items are divided into X disjoint sets and at most one item from each set may be placed in the knapsack. Values are typically assigned to each item and the goal is to maximize the value of the items placed in the knapsack.

MMKP problems can be reduced to optimal feature selection problems with resource constraints. First, a single root feature denoting the solution is created. Next, for each set, a required child feature representing the set is added to the feature model as a child of the root. For each set, the corresponding feature in the feature model is populated with a child XOR group[1] containing the items in the set. The available resources for the feature selection problem are defined as the size of the knapsack. The resources consumed by each feature are assigned to the length, width, and height of the original MMKP item. An example feature model of an MMKP problem is shown in Figure 1.

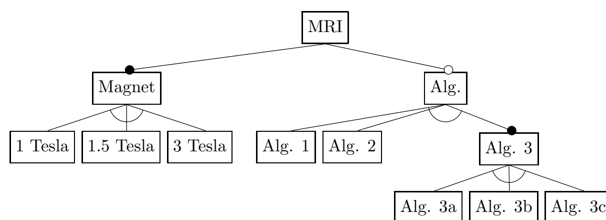


Figure 2: An Example MRI Feature Model

FCF works by performing the reverse process–transforming feature selection problems with resource constraints into MMKP problems. The steps in the FCF technique are designed to flatten the hierarchically structured feature model into a number of independent MMKP sets to form an MMKP problem. Figure 2 shows an example feature model of an MRI machine and Figure 3 illustrates the equivalent MMKP problem. Each item in these sets represents a potential *valid* partial feature selection from the feature model. There are an exponential number of potential feature selections and thus some of the potential configurations must be filtered out to limit the time complexity of the technique. FCF performs this filtering in the third step of the algorithm, described in Section 3.4.

Since each MMKP set that is produced by FCF

[1] An XOR group is a set of features of which exactly one of the features may be selected at a time.

```
MMKP Set 1:
<MRI, Magnet, 1 Tesla>,
<MRI, Magnet, 1.5 Tesla>,
<MRI, Magnet, 3 Tesla>

MMKP Set 2:
<Alg., Alg 1>,
<Alg., Alg 2>,
<Alg., Alg 3, Alg. 3a>,
<Alg., Alg 3, Alg. 3b>,
<Alg., Alg 3, Alg. 3c>

Note:
(items/feature selections denoted with '< >')
```

Figure 3: MMKP Sets for MRI Feature Model

contains items representing valid partial feature selections, the technique must ensure that choosing items from any of the X MMKP sets produces a feature selection that is both valid and complete. The FCF algorithm accomplishes this task in its first step (see Section 3.2) by creating one MMKP set for the subtree of features directly required by the root feature. The remaining MMKP sets are produced from the subtrees of features that are connected to the root through an optional feature. The technique does not currently support cross-tree constraints, although this is part of our future research.

3.2 Step 1: Cutting the Feature Model Graph

The first step in FCF is to subdivide the feature model into a number of independent subtrees. The goal is to choose the subtrees so that the selection of features in one subtree does not affect the selection of features in other subtrees. One MMKP set will later be produced for each subtree.

We define features that are optional or involved in an XOR group or a cardinality group as *choice points*. A cardinality group is a group of features that when selected must adhere to a cardinality expression (*e*.g., select $2 \ldots 3$ of the features X, Y, and Z). An XOR group is a special case of a cardinality group where exactly one feature from the group must be selected (*e*.g., it has cardinality $1 \ldots 1$). Starting from the root, a depth-first search is performed to find each optional feature with no ancestors that are choice points. At each optional feature with no choice point ancestors, a cut is performed to produce a new independent subtree, as shown in Figure 4.

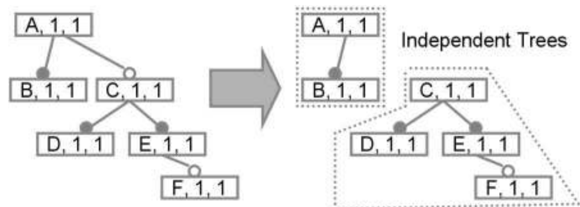


Figure 4: Cutting to Create Independent subtrees

3.3 Step 2: Converting All Feature Constraints to XOR Groups

Each MMKP set forms an XOR group of elements. Since MMKP does not support any other relationship operators, such as cardinality, nor does it support hierarchy, we must flatten each of the subtrees and convert all of their relationship types into XOR. This conversion allows the conversion of the feature model's independent subtrees into a series of MMKP sets.

Cardinality groups are converted to XOR groups by replacing the cardinality group with an XOR group containing all possible combinations of the cardinality group's elements that satisfy the cardinality expression. Each new item produced from the Cartesian product has the combined resource consumption and value of its constituent features. Since this conversion could create an exponential number of elements, we bound the maximum number of elements that are generated to a constant number K. Rather than requiring exponential time, therefore, the conversion can be performed in constant time.

The conversion of cardinality groups is one of the first steps where approximation occurs. We define a filtering operation that chooses which K elements from the possible combinations of the cardinality

group's elements to add to the XOR group. All other elements are discarded.

Any number of potential filtering options can be used. Our experiments evaluated a number of filtering strategies, such as choosing (1) the K highest valued items, (2) a random group of K items, and (3) a group of K items evenly distributed across the items' range of weights. We define a feature's weight or size as the amount of each resource consumed by the feature. We found that selecting the K items with the best ratio of $\frac{Value}{\sqrt{\sum rc_i^2}}$, where rc_i is the amount of the i_{th} resource consumed by the item, provided the best results. This sorting critera has been used successfully by a number of other MMKP algorithms [2]. An example conversion with $K = 3$ and random selection of items is shown in Figure 5.

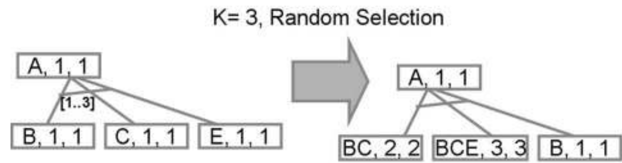


Figure 5: Converting a cardinality group to an XOR Group with K=3 and Random Selection

Individual features with cardinality expressions attached to them are converted to XOR groups using the same method. The feature is considered as a cardinality group containing M copies of the feature, where M is the upper bound on the cardinality expression (*e.g.* $[L..M]$ or $[M]$). The conversion then proceeds identically to cardinality groups.

Optional features are converted to XOR groups by replacing the optional feature O with a new required feature O'. O' in turn, has two child features, O and $\emptyset$ forming an XOR group. O' and $\emptyset$ have zero weight and value. An example conversion is shown in Figure 6.

3.4 Step 3: Flattening with Filtered Cartesian Products

For each independent subtree of features that now only have XOR and required relationships, an

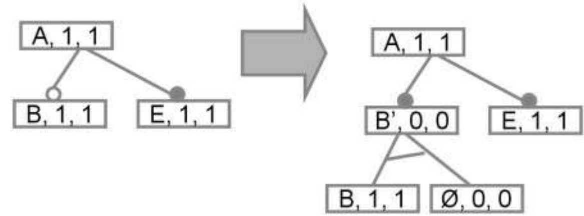


Figure 6: Converting an Optional Feature into an XOR Group

MMKP set needs to be produced. Each MMKP set needs to consist of a single top-level XOR group. To create a single top-level XOR group, we perform a series of recursive flattening steps using filtered Cartesian products to produce an MMKP set containing complete and valid partial feature selections for the subtree.

For each feature with a series of required children, a set of items is produced from a filtered Cartesian product of the sets produced by recursively running the algorithm on its children. For each feature with an XOR group child, a set of items is produced consisting of the Cartesian product of the feature and the union of the sets produced by recursively applying the algorithm to the features in its XOR subgroup. This process is repeated until a single set of items remains. A visualization of this process is shown in Figure 7.

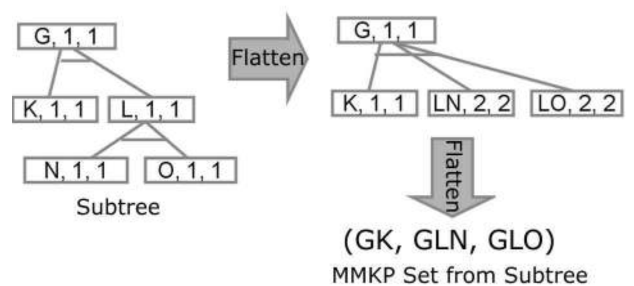


Figure 7: Flattening an XOR Group

Once each independent subtree has been converted into an MMKP set, we must mark those sets which represent optional configuration choices. For each set that does not include the root feature, we add an item

$\emptyset$ with zero weight and zero value indicating that no features in the set are chosen. This standard MMKP method handles situations where choosing an item from some sets is optional. Since the root feature must always be chosen, the $\emptyset$ item is not added to its set.

3.5 Step 4: MMKP Approximation

The first three steps produce an MMKP problem where each set contains items representing potential partial configurations of different parts of the feature model. One set contains partial configurations for the mandatory portions of the feature model connected to the root. The remaining sets contain partial configurations of the optional subtrees of the feature model.

The final step in deriving an optimal architectural feature selection is to run an existing MMKP approximation algorithm to select a group of partial configurations to form the feature selection. For our implementation of FCF, we used a simple modification of the M-HEU algorithm [2] that puts an upper limit on the number of upgrades and downgrades that can be performed. Since FCF produces an MMKP problem, we can use any other MMKP approximation algorithm, such as C-HEU [10]) which uses convex hulls to search the solution space. The solution optimality and solving time will vary depending on the algorithm chosen.

3.6 Algorithmic Complexity

The complexity of FCF's constituent steps can be analyzed as follows, where n is the number of features in the feature model:

- The first step in the FCF algorithm, cutting the tree, requires $O(n)$ time to traverse the tree and find where to make the top-level optional features.
- The second step of the algorithm requires $O(Kn * S)$ steps, where S is the time required to perform the filtering operation. Simple filtering operations, such as random selection, do not add any algorithmic complexity. In these cases, at most n sets of K items must be created to convert the tree to XOR groups, yielding $O(Kn)$. In our experiments, we selected the K items with the best value to resource consumption ratio. With this strategy, the sets must be sorted, yielding $O(Kn^2 \log n)$.
- The third step in the algorithm requires flattening at most n groups using filtered Cartesian products, which yields a total time of $O(Kn{*}S)$.
- The solving step incurs the algorithmic complexity of the MMKP approximation algorithm chosen.

This analysis yields a total general algorithmic complexity for FCF of $(n + (Kn * S) + (Kn * S) + MMKP{+}n) = O(Kn{*}S{+}MMKP{+}n)$. As long as a polynomial time filtering operation is applied, FCF will have an overall polynomial time complexity. For large-scale problems, this polynomial time complexity is significantly better than an exponential running time.

4 Results

To evaluate our FCF approximation technique, we generated random feature models and then created random feature selection problems with resource constraints from the feature models. For example, we would first generate a feature model and then assign each feature an amount of RAM, CPU, etc. that it consumed. Each feature was also associated with a value. We randomly generated a series of available resources values and asked the FCF algorithm to derive the feature selection that maximized the sum of the value attributes while not exceeding the randomly generated available resources. Finally, we compared the FCF answer to the optimum answer.

We performed the experiments using 8 dual processor 2.4ghz Intel Xenon nodes with 2 GB RAM. Each node was loaded with Fedora Core 4. We launched 2 threads on each machine, enabling us to generate and solve 16 optimal feature selection with resource constraints problems in parallel.

The results from solving 18,500 different feature selection problems, each with a feature model of 5,000 features and 2 resource types (RAM and CPU) is shown in Figure 8. We set the max set size, K, in the filtering steps to 2,500. The X axis shows the percentage of optimality. The Y axis shows the number of problem instances or samples that were solved with the given optimality. The overall average optimality was 93%.

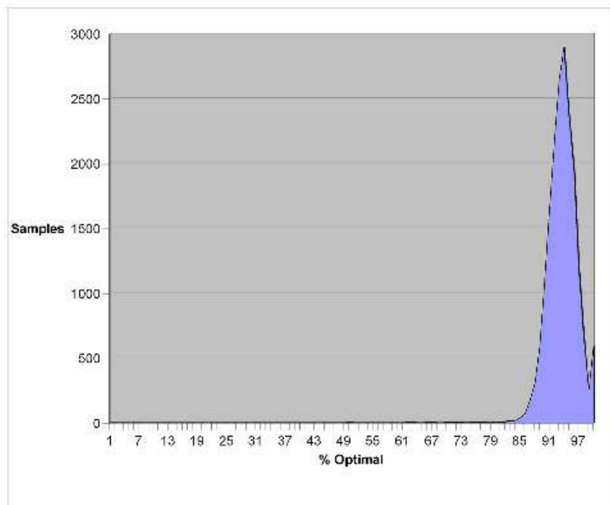


Figure 8: FCF Optimality on 18,500 Feature Models with 5,000 Features and 2 Resources

5 Related Work

Benavides et al. [4] present a technique for using Constraint Satisfaction Problems (CSPs) to model and solve feature selection problems. This technique can be modified to solve feature selection problems subject to resource constraints [13]. Their technique works well for small-scale problems, where an approximation technique is not needed. For larger-scale problems, however, their technique is too computationally demanding. In contrast, FCF performs well on these large-scale problems.

Other approaches to automated feature selection rely on propositional logic, such as those presented by Mannion [8] and Batory [3]. These techniques were not designed to handle integer resource constraints and thus are not equipped to handle optimal feature selection problems subject to resource constraints. Moreover, these techniques rely on SAT solvers that use exponential algorithms. FCF is a polynomial-time algorithm that can handle integer resource constraints and thus can perform optimal feature selection subject to resource constraints on large-scale problems.

6 Conclusion

Approximation algorithms are needed to optimally select features for large-scale feature models subject to resource constraints. Although there are numerous approximation algorithms for other NP-hard problems, they do not directly support optimal feature selection subject to resource constraints. The closest possible class of approximation algorithms that could be applied are MMKP approximation algorithms but these algorithms are not designed to handle the hierarchical structure and non-XOR constraints in a feature model. This lack of approximation algorithms limits the scale of model on which automated feature selection subject to resource constraints can be performed.

This paper shows how the *Filtered Cartesian Flattening* (FCF) approximation technique can be applied to optimally select features subject to resource constraints. FCF creates a series of filtered Cartesian products from a feature model to produce an equivalent MMKP problem. After an equivalent MMKP problem is obtained, existing MMKP approximation algorithms can be used to solve for a feature selection. The empirical results in Section 4 show how FCF can achieve an average of at least 93% optimality for large feature models. The results can be improved by using more exact MMKP approximation algorithms, such as M-HEU [2].

From our experience with FCF, we have learned the following lessons:

- For small-scale feature models (*e.g.*, with < 100 features) MMKP approximation algorithms do

not provide optimal results. For these smaller problems, exact techniques, such as those designed by Benavides et al. [4], should be used.

- For large-scale feature models (*e.g.*, with > 5,000 features) exact techniques typically require days, months, or more, to solve optimal feature selection problems subject to resource constraints. In contrast, FCF typically requires between 1-5 seconds for a 5,000 feature problem.
- Once a feature model has been subdivided into a number of independent subtrees, these subtrees can be distributed across independent processors to flatten and solve in parallel. The FCF technique is thus highly amenable to multi-core processors and parallel computing.

An implementation of FCF is included with the open-source GEMS Model Intelligence project and is available from `eclipse.org/gmt/gems`.

References

[1] M. Akbar, E. Manning, G. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. *International Conference on Computational Science*, pages 659–668, May 2001.

[2] M. Akbar, E. Manning, G. Shoja, and S. Khan. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. pages 659–668. Springer, May 2001.

[3] D. Batory. Feature Models, Grammars, and Prepositional Formulas. *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005: Proceedings*, 2005.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSE05, Proceedings), LNCS*, 3520:491–503, 2005.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[7] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5(0):143–168, January 1998.

[8] M. Mannion. Using first-order logic for product line model validation. *Proceedings of the Second International Conference on Software Product Lines*, 2379:176–187, 2002.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. *Proceedings of the 38th conference on Design automation*, pages 530–535, 2001.

[10] M. Mostofa Akbar, M. Sohel Rahman, M. Kaykobad, E. Manning, and G. Shoja. Solving the Multidimensional Multiple-choice Knapsack Problem by constructing convex hulls. *Computers and Operations Research*, 33(5):1259–1273, 2006.

[11] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 141–150, 2007.

[12] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.

[13] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated Model-based Configuration of Enterprise Java Applications. In *The Enterprise Computing Conference, EDOC*, Annapolis, Maryland USA, October 2007.

Knowledge Based Method to Validate Feature Models

Abdelrahman Osman, Somnuk Phon-Amnuaisuk, Chin Kuan Ho
Center of Artificial Intelligent and Intelligent computing,
Multimedia University, Cyberjaya, Malaysia
abdelrahman.osman.06@mmu.edu.my,somnuk.amnuaisuk@mmu.edu.my,ckho@mmu.edu.my

Abstract

Feature model has been used to support requirements analysis and domain engineering in Software Product Line by representing variability. This paper proposes a knowledge base method to validate feature model. Validation of feature model has been split into two main processes, automated consistency check (by defining rules control the variation selections considering cross-tree constraint dependencies) and the second process is automated error detection(two types of errors, dead feature and inconsistency, was defined and validated). Variability was described among feature model, and then variability was represented as a knowledge base containing predicates and rules. In addition to validation, the proposed method can be used to identified and provide auto support for some operations on the automated analysis of feature models (propagation, cardinality validation, explanation, and optimization).

1 INTRODUCTION

Software Product lines has been defined by Meyer and Lopez as “a set of products that share a common core technology and address a related set of market applications”[1]. *Software product lines* has two main processes; the first process is the domain engineering process which represents problem space and responsible for preparing domain artifacts including *variability*. The second process is the application engineering which aims to consume specific artifact, picking through *variability*, with regard to the desired application specification. One of the useful techniques to represent *variability* is the feature model introduced first in [2]. A particular product-line member is defined by a unique combination of features. The set of all legal feature combinations defines the set of product-line members.

1.1 Feature Models

According to [3], the two most popular definitions of feature models are: i) an end user visible characteristic of a system, and ii) a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept. A *feature model* is a description of the commonalities and differences between the individual software systems in a software product family. In more detail, a *feature model* defines a set of valid feature combinations. Each such valid feature combination can serve as a specification of a software system [4]. Czarnecki defines Cardinality-based feature modeling as integrates a number of extensions to the original FODA notation in [5]. It is very important to producing error-free feature models, including the possibility of providing explanations to the modeler so that errors can be detected and removed; this process is unfeasible to be done manually.

2. Related Work

One of the earlier and famous usages of knowledge based systems in requirement engineering are: "knowledge base software assistant, (KBSA)"[6], and “knowledge Based system for Modeling software system Specifications (KBMS)” [7]. Schlich and Hein proved the needs and benefits of using the knowledge base representation for configuration systems in [8].
Knowledge Acquisition and Sharing for Requirement Engineering (KARE) is a project aims to support Requirement Engineering process with knowledge engineering [9]. A knowledge-based product derivation process [10], [11] is a configuration model that includes three entities of Knowledge Base. The automatic selection provide a solution to complexity of product line variability, but in contrast to our proposed method, the knowledge-based product derivation process does not provide explicit definition of variability notations and for the selection process. In addition to that, knowledge-

based product derivation process is not focused on modeling and validating variability.

Mannion was the first to connect propositional formulas to feature models [12]. Zhang [13] defined a meta-model of feature model using UML core package and he took Mannion's proposal as base and suggested the use of an automated tool support based on the SMV System, Model Checking @CMU. Benavides[14] proposed a method for mapping feature model into constraint solver, his method does not cover dependencies such as require or exclude constraints. Batory [15] proposed a coherent connection between FMs, grammars and propositional formulas; he represented basic FMs used context–free grammars plus propositional logic. Janota [16] used higher-order logic to reasoning about feature models, but there is no real implementation was described. Czarnecki proposed a general template-based approach for mapping feature models in [17]. And he used object- constraint language (OCL) in [18] to validate constraint rules. In contrast to all these models our proposed method defines across-tree constraints between (variation point-variation point), (variation point-variant), and (variant-variant) and can validate cardinality and extended features.

Benavides in [19] presented a survey on the automated analysis of feature models.

Trinidad defined a method to detect dead features based on finding all products and search for features not used in them [20]. He automated error analysis based on theory of diagnosis [21], mapped feature model to diagnose model and used CSP to analyze feature models. Trinidad's method just deals with a dead feature, while our approach deals with three types of errors, inconsistency, dead features, and redundancy. Inconsistency error in feature model was described by Batory [22] as a research challenge. Compare with methods discussed in this literature review this method is first method to deal with inconsistency.

3. Modeling variability using knowledge base rules

A method of modeling variability was proposed to represent domain engineering process as a knowledge base.

First traditional feature model was extended by adding variability notations and modeling domain engineering process as a knowledge base, and then variation point, variants, and constraint dependency rules was represented using predicates. In the next sections we describe extension of feature model using variability notations, how we can automate consistency check using knowledge base rules, and we illustrate how our method can be used to identify and provide auto support for error check and for some operations in feature model analysis process.

3.1. Extending Feature Models by Variability Notations

To maximize the benefits of feature models *variability* notations was characterized from Orthogonal Variability Model (OVM) [23], in feature models as appears in figure 1. Orthogonal Variability Model is one of the useful techniques to represent variability which provides a cross-sectional view of the variability through all software development artifacts. *Variability* - in OVM- is described by variation points, variants and constraint dependencies rules. Constraint dependencies rules as in [24] are: variation point require or exclude variation point, variant require or exclude variant, and variant require or exclude variation point.

Optional and mandatory constraint were defined in figure 1 by original feature model notations [2] ,and constraint dependencies were described using OVM notations.OVM and *feature models* can easily become very complex to model a medium level system, i.e., several thousand of variation points and variants are needed. For this reason we propose an intelligent method to modeling *variability* in *software product line*.

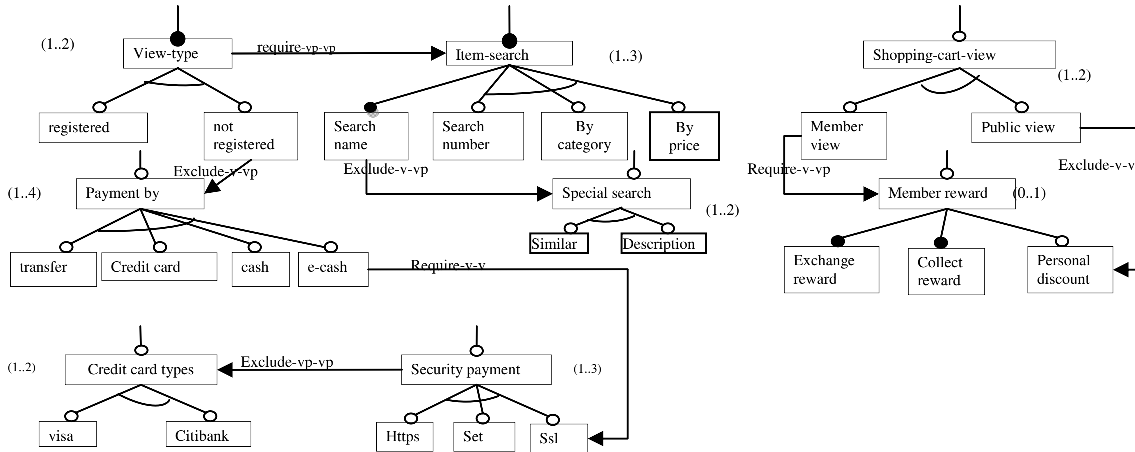


Figure1: Feature model representing variability with e-shopping

Table 1: predicates represent constraint dependency rules in the proposed method.

requires_v_v: Variant requires variant *require_v_v(x,y)| x,y ∈{V}; V= variant*	The selection of a variant *V*1 requires the selection of another variant *V*2 independent of the variation points the variants are associated with. e.g. requires_v_v (ecash, ssl).
excludes_v_v: Variant excludes variant *exclude_v_v(x,y)| x,y ∈{V}; V= variant*	The selection of a variant V1 excludes the selection of the related variant V2 independent of the variation points the variants are associated with .e.g. excludes_v_v(public_view, personal_discount).
requires_v_vp: Variant requires variation point *require_v_vp(x,y)| x,y ∈{V,VP}; V= variant.Vp=variation point*	The selection of a variant V1 requires the consideration of a variation point VP2. *e.g.* requires_v_vp (member_view, member_reword).
excludes_v_vp: Variant excludes variation point *excludes_v_vp(x,y)|x,y ∈{V,VP};V=variant.VP= variation point*	The selection of a variant V1 excludes the consideration of a variation point VP2. *e.g.* excludes_v_vp (not registered, payment_by).
requires_vp_vp: Variation point requires variation point *require_vp_vp(x,y) | x,y ∈{VP}; VP= variation point*	A variation point requires the consideration of another variation point in order to be realized. *e.g.* requires_vp_vp (item_search, view_type).
excludes_vp_vp Variation point excludes variation point *excludes_vp_vp(x,y)|x,y∈{VP}; VP= variation point*	The consideration of a variation point excludes the consideration of another variation point. *.e.g.* excludes_vp_vp (security_payment, credit_card_type).

3.2. Variation points and variants as predicates

In this section variation point and variant were described using predicates: (examples are based on figure 1)

- *type*: Describes the type of feature; variation point or variant. *e.g.*: type (view_type, variationpoint), type (register, variant).
- variant: Identifies the variant of specific variation point. *e.g.*: variant(view_type, not registered)
- *max*: Identifies the maximum number allowed to be selecting of specific variation point. *e.g.* max (payment_by, 4).
- *min*: Identifies the minimum number allowed to be selecting of specific variation point. *e.g.* min (payment_by, 1).

The common variant(s) in a variation point is/are not included in maximum-minimum numbers of selection.

- *common*: describe the commonality of specific feature. *e.g.* common (search_name, yes). If the feature is not common, the second slot in the predicate will become No -as example- common (register, no).

Predicates were used to represent constraint dependency rules between features. Table 1 describes predicates represent constraint dependency rules with examples from figure 1.

Table 2 describe the proposed predicates using Backus–Naur form (BNF).

Table 2: Representation of variant and variation point using Backus–Naur form (BNF).

<type > ::= <variationpoint> |< variant>
<variationpoint>::=[<name><cardinality><variant> <common>]
<name> ::= <String>
<cardinality> ::= [<min> <max>]
<min> ::= <Digits>
<max> ::= <Digits>
<variant> ::= <variant> <variant>
<variant> ::= [<name> <common>]
<common> ::= <Yes> | <No>

Table 3 shows the representation of *view-type* variation point from figure 1. Table 4 shows the representation of *not registered* variant from figure 1.

Table 3: view-type representation

type (view-type, variationpoint).
variants (view-type, registered).
variants (view-type, not registered).
common (view-type, yes).
min (view-type, 1).
max (view-type, 3).
requires_vp_vp(view-type,search_item).

Table 4: not registered representation

type (not registered, variant).
common(not registered, no).
excludes_v_vp(not registered, payment by).

Table 5: Abstract representation of the main rules in the proposed model

1. $\forall x, y: type(x, variant) \land type(y, variant) \land require_v_v(x, y) \land select(x) \Rightarrow select(y)$
2. $\forall x, y: type(x, variant) \land type(y, variant) \land exclude_v_v(x ,y) \land select(x) \Rightarrow notselect(y)$
3. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land require_v_vp(x, y) \land select(x) \Rightarrow select(y)$
4. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land exclude_v_vp(x, y) \land select(x) \Rightarrow notselect(y)$
5. $\forall x, y: type(x, variationpoint) \land type(y, variationpoint) \land require_vp_vp(x, y) \land select(x) \Rightarrow select(y)$
6. $\forall x, y: type(x, variationpoint) \land type(y, variationpoint) \land exclude_vp_vp(x, y) \land select(x) \Rightarrow notselect(y)$
7. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land select(x) \land variants(y, x) \Rightarrow select(y)$
8. $\exists x \ \forall y: type(x, variant) \land type(y, variationpoint) \land select(y) \land variants(y, x) \Rightarrow select(x)$
9. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land notselect(y) \land variants(y, x) \Rightarrow notselect(x)$
10. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land common(x,yes) \land variants(y, x) \land select(y) \Rightarrow select(x)$
11. $\forall y: type(y, variationpoint) \land common(y,yes) \Rightarrow select(y)$
12. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land variants(y, x) \land select(x) \Rightarrow sum(y,(x)) \leq max(y,z)$
13. $\forall x, y: type(x, variant) \land type(y, variationpoint) \land variants(y, x) \land select(x) \Rightarrow sum(y,(x)) \geq min(y,z)$

3.3 Validation Rules

To validate the selection process, our proposed method triggers rules based on constraint dependencies. With regard to validation process result, the choice is added to knowledge base or rejected, then an explanation of rejection reason is provided and correction actions suggested. At any new solution generated, new fact (select or notselect) added to the knowledge base and backtracking mechanism validates all. At the end of the process all *select* facts represent the product. Table 5 shows the abstract representation of the main rules in the knowledge base. The proposed method contains 13 main rules to validate the selection process based on constraint dependencies.

Rule 1:
For all variant *x* and variant *y*; if *x* requires *y* and *x* is selected, then *y* should be selected.

Rule2:
For all variant *x* and variant *y*; if *x* excludes *y* and *x* is selected, then *y* should not be selected.

Rule 3:
For all variant *x* and variation point *y*; if *x* requires *y* and *x* is selected, then *y* should be selected.
This rule is applicable as well if *y* selected with the same condition:
$\forall x, y: type(x, variant) \land type(y, variationpoint) \land require_v_vp(x, y) \land select(y) \Rightarrow select(x)$
For all variant *x* and variation point *y*; if *x* requires *y* and *y* is selected, then *x* should be selected.

Rule 4:
For all variant *x* and variation point *y*; if *x* excludes *y* and *x* is selected, then *y* should not be selected.
This rule is applicable as well if *y* selected with the same condition:
$\forall x, y: type(x, variant) \land type(y, variationpoint) \land exclude_v_vp(x, y) \land select(y) \Rightarrow notselect(x)$
For all variant *x* and variation point *y*; if *x* excludes *y* and *y* selected, then *x* should not be selected.

Rule 5:
For all variation point *x* and variation point *y*, if *x* requires *y* and *x* selected, then *y* should be selected.

Rule 6:
For all variation point *x* and variation point *y*, if *x* excludes *y* and *x is* selected, then *y* should not be selected.
Rule 7:
For all variant *x* and variation point *y*, where *x* belongs to *y* and *x* is selected, that means *y* should be selected.
This rule determines the selection of variation point if one of its variants was selected.
Rule 8:
For all variation point *y* there exists of variant *x*, if *y* selected and *x* belongs to *y*, *x* should be selected.
This rule states that if a variation point was selected, then there is variant(s) belong to this variation point should be selected.
Rule 9:
For all variant *x* and variation point *y*; where *x* belongs to *y* and *y* defined by predicate *notselect(y)*, then *x* should not be selected.
This rule states that if a variation point was excluded, then none of its variants should be selected.
Rule 10:
For all variant *x* and variation point *y*; where *x* is a common and *x* belongs to *y* and *y* is selected, then *x* should be selected.
This rule states that if a variant is common and its variation point selected then it must be selected.
Rule 11:
For all variation point *y*; if *y* is common, then *y* should be selected.
This rule states that if a variation point is common then it should be selected in any product.
Rule 12:
For all variant *x* and variation point *y*; where x belongs to *y* and *x* is selected, then the summation of *x* should not be less than the maximum number allowed to be selected from *y*.
Rule 13:
For all variant *x* and variation point *y*; where *x* belongs to *y* and *x* is selected, then the summation of *x* should not be greater than the minimum number allowed to be selected from *y*.
Rules 12 and 13 validate the number of variants' selection considering the maximum and minimum conditions in variation point definition. The predicate *sum(y, (x))* return the summation number of selected variants belong to variation point *y*.

From these rules we can define a full common variant, variant included in any product, as:

$\forall x,y{:}type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge common(y,yes) \wedge common(x,yes) \Rightarrow full_common(x)$

A full common variant is a common variant belongs to common variation point. A common variation point included in any product (rule 11), a common variant belongs to selected variation point should be selected (rule 10).

These rules were implemented using SWI-Prolog [25]. In real implementation we have numerous rules to cover all situations, but all the rules based on the 13 main rules. Definitions and examples were described in the next section (based on figure1) to illustrate the usefulness of our proposed method of modeling variability.

3.4. Operations on the Automated Analysis of Feature Models:

The main operations of automated analysis of feature models defined in [19]. Our proposed method can define and provide auto support for a number of operations (feature Model validation, propagation, explanation, optimization, dead feature detection, inconsistency detection, and cardinality validation).

3.4.1. Propagation. This operation returns a feature model where some features are automatically selected (or deselected).

Definition 1
Selection of variant *n*, *select(n)*, is propagated from selection of variant *x*, *select(x)*, in three cases:

i. $\forall x,y,z,n{:}type(x,variant) \wedge variants(y,x) \wedge select(x) \wedge requires_vp_vp(y,z) \wedge type(n,variant) \wedge variants(z,n) \wedge common(n,yes) \Rightarrow select(n)$.

If *x* is a variant and *x* belongs to the variation point *y* and *x* is selected, that means *y* is selected (rule 7), and the variation point *y* requires a variation point *z*, that means *z* is selected also (rule 5), and the variant *n* belongs to the variation point *z* and the variant *n* is common that means the variant *n* is selected (rule 10).

ii. $\forall x,n{:}\ type(x,variant) \wedge type(n,variant) \wedge select(x) \wedge requires_v_v(x,n) \Rightarrow select(n)$.

If the variant *x* is selected and it requires the variant *n*, that means the variant *n* is selected, (rule 1). The selection of variant *n* propagated from the selection of variant *x*.

iii. $\forall x,z,n{:}type(x,variant) \wedge select(x) \wedge type(z,variationpoint) \wedge requires_v_vp(x,z) \wedge type(n,variant) \wedge variants(z,n) \wedge common(n,yes) \Rightarrow select(n)$.

If the variant *x* is selected and it requires the variation point z that means the variation point *z* is selected (rule 3), and the variant *n* is common and is belongs to the variation point *z* that means the variant *n* is selected (rule 10). The selection of variant *n* propagated from the selection of variant *x*.

Example 1
Suppose the user entered this choice *select (register)*, the system answered *yes* (acceptance of user selection) user announced by selection of the variant *search_name*, as propagated from selection of the variant *register.*

Table 6: example 1

?select (view_type.register). yes You selected also…. search_name

This example illustrates case 1. *view_type* variation point requires *item_serach* variation point and *search_name* is mandatory variant belongs to the variation point *item_search* . The direct selection of variant *register* makes *view_type* variation point selected (rule 7), and the selection of *view_type* variation point makes the *item search* variation point selected (rule 5), then the common variant *search_name(* belongs to *item_search* variation point) should be selected (rule 10). The main result of this example is the additions of two new facts select *(register)* and *select (search_name)* to knowledge base.

3.4.2. Explanation. This operation takes a feature model as an input and returns an explanation in the case when the feature model fails. Definition of error source (the constraint dependency rule(s) that causing the error) is the main aim of explanation in feature models.

Definition 2

Selection of variant *n*, *select (n)*, fails due to selection of variant *x*, *select(x)*, in three cases:

i. $\forall x,y,n: type(x,variant) \wedge select(x) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(n,variant) \wedge excludes_v_vp(n,y) \Longrightarrow notselect(n).$

If the variant *x* is selected, and it belongs to the variation point *y*, which means *y* is selected (rule 7), and the variant *n* excludes the variation point *y*, which means *n* should not be selected (rule 4 is applied also if the variation point is selected).

ii. $\forall x,y,z,n: type(x,variant) \wedge select(x) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge variants(z,n) \quad excludes_vp_vp(y,z) \Longrightarrow notselect(n).$

If the variant *x* is selected and *x* belongs to the variation point *y*, that means *y* is selected (rule 7), and the variation point *y* excludes the variation point *z*, that means *z* should not be selected (rule 6), and the variant *n* belongs to variation point *z*, that means *n* should not be selected same wise (rule 9).

iii. $\forall x,n \quad type(x,variant) \wedge select(x) \wedge type(n,variant) \wedge excludes_v_v(x,n) \Longrightarrow notselect(n).$

If the variant *x* is selected, and it excludes the variant *n*, which means *n* should not be selected(rule 2).

In addition to defining the source of error, these rules can be used to prevent the errors. The predicate *notselect(n)* validate users by prevent selection.

Table 7: example 2

? select (personal_discount). Reject You have to deselect public_view first

Example 2

Suppose user selected *public_view* before and entered new selection, and asks to select *personal_discount*, the system rejects his choice and directed him to deselect *public_view* first. Table 7 describes example 2, this example represents case 3. The example illustrates how the model guides users to solve the rejection reason.

In addition to that the proposed method can be used to prevent rejection reasons; example 3 explains this.

Table 8: example 3

? select (Http). Yes notselect (credit_card_types) added to knowledge base.

Example 3

User asks to select the variant *https*, system accept his choice and add *notselect(credit_card_types)* to the knowledge base to validate future selections. Table 8 describes example 3.

Selection of the variant *Http* from *security_payment* variation point leads to the selection of *security_payment* (rule 7), and *security_payment* excludes *credit_card_types* variation point, which means *credit_card_types* should not be selected (rule 6).The predicate *notselect(credit_card_types)* prevents the selection of its variants according to rule 9.

3.4.3. Optimization. This operation returns the output, product, according to a given function or predefined criteria.

One of the advantages of our proposed method is that it can handle the extra-functional features proposed in [14], we can use extra-functional feature to optimize the search or to make filtering.

Example 4

Suppose we defined price as extra-functional feature to *security_payment* variation point in figure 1, as a result we have new three facts *price(http,100)*, *price(ssl,200)*, and *price(set,350)*. We want to ask about the feature with price greater than 100 and less than 250(price(X, Y), Y > 100, Y< 250), the system triggers the variant *ssl* with price 200. Table 9 describes example 4.

Table 9: Example 4

? price(X, Y), Y > 100, Y< 250. X = ssl Y = 200

3.4.4. Dead Feature Detection. A dead feature is a feature that never appears in any legal product of a feature model, defined [26] as a frequent case of error in feature model.

Definition 3

A variant *x* can be a dead feature in 3 cases:

i. $\forall x,y,z,n: type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variant) \wedge type(n,variationpoint) \wedge variants(n,z) \wedge common(n,yes) \wedge common(z,yes) \wedge excludes_v_vp(z,y) \Longrightarrow dead_feature(x).$

If variant *x* belongs to the variation point *y*, and the common variant *z* belongs to the common variation point

n, which means the variant *z* included in any product (rules 11 and 10), the variant *z* excludes the variation point *y* that means none of its variants should be selected at all(rule 4).

ii. $\forall x,y,z: type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge common(z,yes) \wedge excludes_vp_vp(z,y) \Longrightarrow dead_feature(x).$

If the variant *x* belongs to the variation point *y* and the common variation point *z* excludes the variation point *y* that means *y* should be excluded (rule 6). Exclusion of the variation point *y* prevents selection of its variants (rule 9).

iii. $\forall x,y,n: type(n,variant) \wedge type(y,variationpoint) \wedge variants(y,n) \wedge common(y,yes) \wedge common(n,yes) \wedge type(x,variant) \wedge excludes_v_v(n,x) \Longrightarrow dead_feature(x).$

If the common variant *n* belongs to the common variation point *y* that means *y* should be selected in any product (rule 11), when y is selected n should be selected (rule 10), which means *n* should be selected in any product. The variant *n* excludes the variant *x* that means *x* should not be selected in any product (rule 2).

Example 5

In figure1 the variant *search_name* is a common feature from a common variation point *item_search* that means it is included in any product(rules 11 and 10), and it excludes the variation point *special_search* which means none of its variants should be selected in any product (rule 9). This means that all variants belonging to the variation point *special_ search* are dead features. Table 10 describes example 5, user inquired about dead features and system triggered the variants *similar* and *description* as dead features, this example represents case 1.

Table 10: example 5

```
? dead_feature(X).
X = similar
X = Description
```

Trinidad proposed in [20] a method to detect dead features based on finding all products and search for not used features. Our proposed method to detect dead features is better because it searches only in the above three cases.

3.4.5. Inconsistency. Inconsistency in feature model is a relationship between features that cannot be true at the same time. e.g. (A require B) and (B exclude A)

Definition 4

The inconsistency (error) can be detected in five cases:

i. $\forall x,y: type(x,variant) \wedge type(y,variant) \wedge requires_v_v(x,y) \wedge excludes_v_v(y,x) \Longrightarrow error.$

If the variant *x* requires the variant *y* that means selection of *x* leads to selection of *y* (rule 1). And the variant *y* excludes the variant *x* that means if *y* selected, *x* should not be selected (rule 2), this is an error.

ii. $\forall x,y: type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires_vp_vp(x,y) \wedge excludes_vp_vp(y,x) \Longrightarrow error.$

If the variation point *x* requires the variation point *y* that means selection of *x* leads to selection of *y* (rule 5), and the variation point *y* excludes the variation point *x* means if *y,* selected *x* should not be selected (rule 6), this is an error.

iii. $\forall x,y,n,z: type(x,variant) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(n,variant) \wedge type(z,variationpoint) \wedge variants(z,n) \wedge requires_v_v(x,n) \wedge excludes_vp_vp(y,z) \Longrightarrow error.$

If the variant *x* belongs to the variation point *y*, and the variant *n* belongs to the variation point *z*, and *x* requires *n* that means if *x* selected *n* should be selected (rule1). Selection of the variant *x* means selection of the variation point *y*, and selection of variant *n* means selection of variation point *z* (rule 7). The variation point *y* excludes the variation point *z* that means if one of the variants belongs to *y* is selected none belongs to *z* should be selected (rules 6, 7, and 9), this is an error.

iv. $\forall x,y,z: type(x,variant) \wedge common(x,yes) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge excludes_v_vp(x,z) \wedge requires_vp_vp(y,z) \Longrightarrow error.$

If the common variant *x* belongs to the variation point *y*, and *x* excludes the variation point *z* that means if *x* selected no variant belongs to *z* should be selected (rules 4, and rule 9), and the variation point *y* requires the variation point *z* that means if *y* is selected *z* should also be selected(rule 5). Selection of the variation point *y* means selection of the common variant *x* (rule 10) but *x* excludes *z*, this is an error.

v. $\forall x,y,z: type(x,variant) \wedge common(x,yes) \wedge type(y,variationpoint) \wedge variants(y,x) \wedge type(z,variationpoint) \wedge requires_v_vp(x,z) \wedge excludes_vp_vp(y,z) \Longrightarrow error.$

If the common variant *x* belongs to the variation point *y*, selection of *x* means selection of y (rule 7), and *x* requires the variation point *z* that means selection of *x* leads to selection of *z* (rule 3); but the variation point *y* excludes the variation point *z* which means if *y* is selected *z* should not be selected (rule 6), this is an error.

Feature models can contain some other complicated forms of inconsistencies like (A requires B) and (B requires C) and (C excludes A). To avoid this complication the following rules were defined:

i. $\forall x,y,z: type(x,variant) \wedge type(y,variant) \wedge requires_v_v(x,y) \wedge type(z,variant) \wedge requires_v_v(y,z) \Longrightarrow requires_v_v(x,z).$

If the variant *x* requires the variant *y*, and the variant *y* requires the variant *z* that means the variant *x* requires the variant *z*.

ii. $\forall x,y,z: type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires_vp_vp(x,y) \wedge type(z,variationpoint) \wedge requires_vp_vp(y,z) \Longrightarrow requires_vp_vp(x,z).$

If the variationpoint *x* requires the variation point *y*, and the variation point *y* requires the variation point *z* that means *x* requires *z*.

iii. $\forall x,y,z: type(x,variant) \wedge type(y,variationpoint) \wedge requires_v_vp(x,y) \wedge type(z,variationpoint) \wedge requires_vp_vp(y,z)$

$\Rightarrow$ *requires_v_vp(x,z).*

If the variant *x* requires the variation point *y*, and *y* requires the variation point *z* that means the variant *x* requires the variation point *z*.

iv. $\forall x,y,z$*:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*requires_v_v(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*requires_v_vp(y,z)*
$\Rightarrow$ *requires_v_vp(x,z).*

If the variant *x* requires the variant *y* and *y* requires the variation point *z* that means the variant *x* requires the variation point *z*.

v. $\forall x,y,z$*:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*excludes_v_vp(x,y)*$\wedge$*type(z,variant)*$\wedge$*excludes_v_v(y,z)* $\Rightarrow$ *excludes_v_v(x,z).*

If the variant *x* excludes the variant *y* and the variant *y* excludes the variant *z* that means the variant *x* excludes the variant z.

vi. $\forall x,y,z$*:type(x,variationpoint)*$\wedge$*type(y,variationpoint)*$\wedge$*excludes_vp_vp(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_vp_vp(y,z)* $\Rightarrow$ *excludes_vp_vp(x,z).*

If the variationpoint *x* excludes the variationpoint *y*, and the variation point *y* excludes the variation point *z* that means *x* excludes *z*.

vii. $\forall x,y,z$*:type(x,variant)*$\wedge$*type(y,variationpoint)*$\wedge$*excludes_v_vp(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_vp_vp(y,z)*
$\Rightarrow$ *excludes_v_vp(x,z).*

If the variant *x* excludes the variationpoint *y*, and the variation point *y* excludes the variation point *z* that means *x* excludes *z*.

viii. $\forall x,y,z$*:type(x,variant)*$\wedge$*type(y,variant)*$\wedge$*excludes_v_v(x,y)*$\wedge$*type(z,variationpoint)*$\wedge$*excludes_v_vp(y,z)*
$\Rightarrow$ *excludes_v_vp(x,z).*

If the variant *x* excludes the variant *y*, and *y* excludes the variation point *z* that means the variant *x* excludes the variation point *z*. Using backtracking mechanism the above rules can solve more complex shapes of inconsistency such as ((A requires B) and (B requires C) and (C requires D) and (D requires F) and (F excludes A)).

Our proposed method to auto detected error in feature model is better than that suggested by Trinidad (P. Trinidad 2008) because it defines two types of error in feature models.

3.4.6. Cardinality. Cardinality of variation point represents the minimum and maximum number of variant selection.

Definition 5

We define cardinality using two predicates:

i. *min(name,num)* represent the minimum cardinality; *name* represents variation point name, and *num* is an integer represents the minimum number allowed to be selected from variation point described in *name*.
ii. *max(name,num)*represent the maximum cardinality; *name* represents variation point name, and *num* is an integer represents the maximum number allowed to be selected from variation point described in *name*. Rule 12, and 13 validate cardinality.

4. Discussions and Comparison with Previous Work

In addition to automated consistency check among constraints during modeling, our proposed method can be considered as a validation and analysis method within modeling. Our proposed method can substantially define and provide an automated mechanism to the following operations:

- Definition of propagation states: identify when the propagation can happen.
- Provide explanation: We identified when the feature model fails to answer users, and we identified all features' relations those lead to feature model failure, i.e. source of error. In addition to that we illustrated how the method can guide users in correction process, in comparison with literature our method is a unique method which can provide correction process; besides all these we defined a validation predicate - *unselect(x)*- this predicate can prevent users from errors. We illustrated how the feature model can be optimized using predicates; the method can deal with the extra features.
- All cases of the dead features were defined using rules: To detect dead features search only for the corresponding defined cases.
- All cases of inconsistency were represented in feature mode using rules: provide a method to detect inconsistency in a feature model is a novel. In addition to that we used rules to define complex types of inconsistency and illustrated that how our method can prevent and solve these types of inconsistencies, which also considered as a core contribution.

5. Conclusion and future work

By modeling variability using knowledge base rules we can get both formalized variability specifications, and support selection process within variability more precisely. Firstly we proposed a method of modeling variability in feature models based product line. We started by used notations of orthogonal variability model to define variability in basic feature model, and then developed knowledge base. Our proposed method of modeling variability aims to improve the effort of producing fully automated product derivation and to deal with the complexity of variability in product line approach (intractability). The proposed knowledge base can be used to configure new product from SPL,

analyzing SPL and produce error free feature model. Russell defined five steps to build knowledge [27], as we defined the five steps therefore our work can be considered as knowledge base of variability in domain engineering process.

We are planning to extend our work using constraint handling rules (CHR) to calculate and obtain all products, calculate variability (the ratio between all product and all variants in the model) , and calculate commonality. Also we are planning to develop software tool to support our method. Finally validate our method by applying it to real life case studies from industry.

6. References

[1] Meyer, M. H., and Luis Lopez, "Technology Strategy in a Software Products Company". *Product Innovation Management,* Blackwell Publishing, vol *12*, 1995, 294-306.

[2] K. Kang, S. C., J. Hess, W. Novak, and S. Peterson, "Feature–Oriented Domain Analysis (FODA) *Feasibility Study*" (Technical Report No. CMU/SEI-90-TR-21) Software Engineering Institute, Carnegie Mellon University,1990.

[3] Krzysztof Czarnecki, U. Eisenecker , *Generative Programming: Methds, Tools, and Applications*, Addison-Wesley, Boston MA, 2000.

[4] Timo Asikainen , T. Männistö, T. Soininen , *"Representing Feature Models of Software Product Families Using a Configuration Ontology"*, Paper presented at the General European conference on artificial intelligence (ECAI)Workshop on Configuration , 2004.

[5] Krzysztof Czarnecki, S. Helsen, U. Eisenecker, *"Staged configuration using feature models",* Paper presented at the Third International Conference of Software Product Lines, SPLC 2004, Boston MA, USA. 2004.

[6] White D. A., *"The Knowledge Base Software Assistant: A program Summary",* Paper presented at the Knowledge-Based Software Engineering Conference, New York USA,1991

[7] kacem Zeroual , P.N. Robillard, *"KBMS: A Knowledge Based System for Modeling Software System Specification"*, IEEE Transactions on Knowledge and Data Engineering, *4*(3),1992 , 238 – 252.

[8] Michael Schlick, A. Hein, *"Knowledge engineering in software product lines",* Paper presented at the 14th European conference on Artificial Intelligent, Workshop on Knowledge-Based Systems for Model-Based Engineering, 2000.

[9] S. Ratchev , E. Urwin, D. Muller , K.S. Pawar, and I. Moulek,*"Knowledge Based Requirement Engineering for one-of-a-kind Complex Systems*", Knowledge base systems ,Elsevier, *16*(1), 2003,1-5.

[10] Lother Hotez , T. Krebs, *"Supporting the Product Derivation Process with a Knowledge Base Approach",* Paper presented at the 25th International Conference on Software Engineering (ICSE2003), 2003.

[11] Lother Hotez , T. Krebs, *"A knowledge based product derivation process and some idea how to integrate product development",* Paper presented at the Software Variability Management Workshop, Groningen The Netherlands, 2003.

[12] M. Mannion , *"Using First-Order Logic for Product Line Model Validation",* Paper presented at the Second Software Product Line Conference (SPLC2), San Diego, CA. , 2002.

[13] Wei Zhang, H. Z., and Hong Mei, *"A Propositional Logic-Based Method for Verification of Feature Models",* Paper presented at the 6th International Conference on Formal Engineering Methods (ICFEM),2004.

[14] David Benavides, P. Trinidad, and A.Ruiz-Cortes, *"Automated Reasoning on Feature Models*", Advanced Information Systems Engineering (Vol. 3520/2005,), Springer, Berlin Heidelberg, 2005, pp. 491-503.

[15] Don Batory,*"Feature Models, Grammars, and Propositional Formulas",* Paper presented at the 9th International Software Product Lines Conference (SPLC05), Rennes, France, 2005.

[16] Mikolas Janota, Joseph Kiniry, *"Reasoning about Feature Models in Higher-Order Logic",* Paper presented at the 11th International Software Product Line Conference (SPLC07), 2007.

[17] Krzysztof Czarnecki, Michal Antkiewicz, *"Mapping features to models: A template approach based on superimposed variants",* Paper presented at the 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, 2005.

[18] Krzysztof Czarnecki, Krzysztof Pietroszek, *"Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints"*, Paper presented at the 5th international conference on Generative programming and component engineering (GPCE'06), 2006.

[19] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura, *"A survey on the automated analyses of feature models"*, *Jornadas de Ingeniería del Software y Bases de Datos(JISBD 2006)*, 2006.

[20] Pablo Trinidad, David Benavides, and Antonio Ruiz-Cort´es, *"Isolated features detection in feature models"*, Paper presented at the Advanced Information Systems Engineering (CAiSE), Luxembour, 2006.

[21] Pablo Trinidad , D. Benavides, A. Dura´n, A. Ruiz-Cortes, and M. Toro, *"Automated error analysis for the agilization of feature modeling"*, systems and software, doi:10.1016/j.jss.2007.10.030, 2008.

[22] Don Batory, David Benavides, Antonio Ruiz-Cortés, *"Automated Analyses of Feature Models: Challenges Ahead"*, Special Issue on Software Product Lines ,Communications of the ACM, December 2006.

[23] Timo Käkölä, Juan C. Dueñas , *Software Product Lines Research Issues in Engineering and Management*, Springer, Verlag Heidelberg Germany, 2006.

[24] Klaus Pohl, Günter Böckle, and Frank van der Linden, *Software Product Line Engineering Foundations Principles and Techniques*, Springer, Verlag Heidelberg Germany, 2005.

[25] Jan Wielemaker ,SWI-Prolog (Version 5.6.36) free software, Amsterdam, University of Amsterdam,2007.

[26] Krzysztof Czarnecki, Chang Hwan Peter Kim, *"Cardinality-based Feature Modeling and Constraints: A Progress Report"*, Paper presented at the International Workshop on Software Factories at (OOPSLA'05), San Diego California, 2005.

[27] Stuart J. Russell , Peter Norvig , *Artificial Intelligence A Modern Approach* , Prentice Hall, New Jersey 07632, 1995.

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.

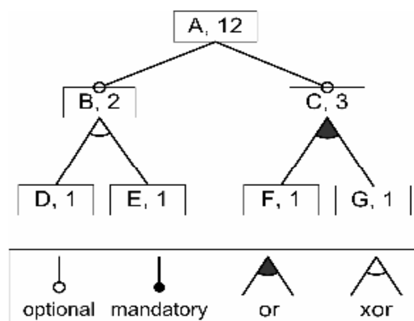


Figure 1. Example feature tree

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:

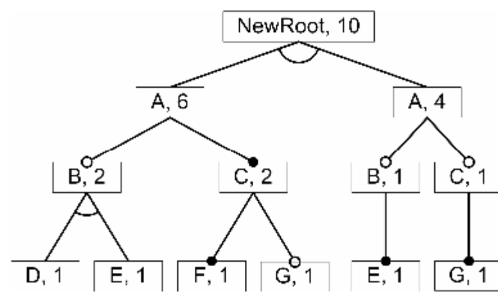


Figure 2. Example feature tree,
constraint "D requires F" eliminated

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:

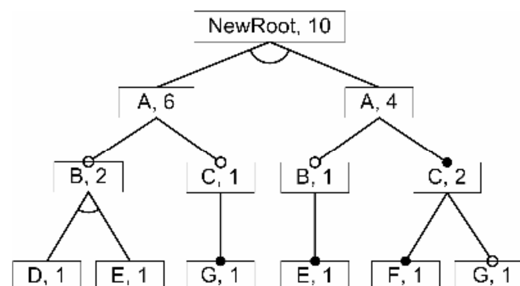


Figure 3. Example feature tree,
constraint "D excludes F" eliminated

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

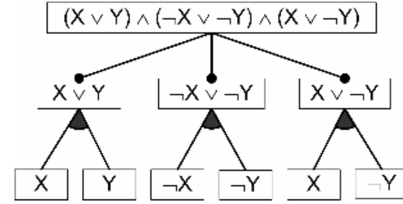


Figure 4

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
```

Understanding Decision-Oriented Variability Modelling

Deepak Dhungana Paul Grünbacher
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz, Austria
{dhungana | gruenbacher}@ase.jku.at

Abstract

Researchers and practitioners have been developing a wide range of techniques and tools to model and manage variability as a response to the heterogeneity of application areas and the diversity of implementation practices in different domains. In our own research we have been developing a tool-supported approach to decision-oriented variability modelling, which is highly customizable to domain-specific needs. In the past we have reported on our experiences on using the approach and its benefits in diverse industrial contexts. In this paper we present a more formal description of our approach and define the execution semantics of decision-oriented variability models.

1. Introduction

Variability is an emergent property of software systems and results from different design decisions taken to address requirements and contexts from different users. Experience from large-scale long-living systems shows that knowledge about variability is mostly tacit in nature and manifests itself in many different kinds of artefacts (documents, software components, test cases, configuration parameters, etc.) and different mechanisms supported by programming languages, architectural styles, design patterns, etc. Variability models have been proposed as a means of communication to deal with explicit documentation of tacit knowledge and better utilization of the flexibility and adaptability provided by a system.

The importance of variability in software systems and the necessity of making knowledge about variability explicit in models have already been identified as important research areas in software engineering. Depending on the background of different researchers, the needs of different industrial contexts and the kinds of systems under investigation, several variability modelling tools and techniques are already available [1, 5, 14, 15, 18, 19]. However, due to the broad spectrum of application areas and the diversity of implementation practices in different domains, a "standard approach" for dealing with variability will probably never exist. There are a lot of "island solutions" for variability modelling which either focus one particular level of abstraction or are monolithic and fixed to a certain grammar, with a set of predefined features. This hinders the widespread use of the existing approaches in different domains and application contexts. Despite the importance of variability modelling and the usage of such models in a wide array of contexts, researchers and practitioners are still struggling to find tools and techniques that best suit their modelling needs.

Feature modelling is probably the most prominent approach for modelling variability. Starting from FODA [13], the feature-oriented view of the world has already gone far beyond variability modelling and system documentation. Several formal interpretations (e.g. survey in [19]) of feature models and their applications have already been published. Today several variants of feature-based variability modelling tools and techniques are available.

A comparably smaller number of publications propose decision-oriented approaches to modelling variability. The idea of decision modelling in product lines is not new; it was introduced by Campbell et. al. [4, 2] in the early 1990s, where decisions were "*actions which can be taken by application engineers to resolve the variations for a work product of a system in the domain*" [2]. Forster et. al. [10], Schmid et. al. [18], Sellier et. al. [20] and others have been actively publishing their research results in this area. Astonishingly, researchers have not yet found a common basis on the notion of decisions. Some researchers follow decision modelling on a rather informal basis (e.g. using tables [18]), while others have already automated the decision

making procedure by using executable descriptions and formal approaches.

We have incorporated a decision-oriented approach into our tool suite DOPLER [8] consisting of the tools DecisionKing [7, 6], ProjectKing [17] and ConfigurationWizard [16]. Here we describe decision models used in DOPLER tools more formally based on our experiences and feedback from industry. We provide a definition of the decision-oriented variability modelling language DoVML.

2. The basics of DoVML

Modelling the variability of software systems involves modelling the problem space (i.e., stakeholder needs or desired features) and the solution space (i.e., the architecture and the components of the technical solution). Separation of concerns based on problem space and solution space was also dealt with by Metzger et. al [14]. Our decision-oriented variability modelling language (DoVML) supports the modelling of the problem space using *decisions* and the solution space using *assets*.

The basic constructs for modelling variability using DoVML are depicted in figure 1. A Variability model is a set of *decisions*, *assets* and *rules*. Decisions can be organized in *groups*. The dependencies between decisions are expressed using *visibility conditions* and *validity conditions*. The dependencies among assets and decisions are established using *inclusion conditions*. *Visibility conditions*, *validity conditions*, and *inclusion conditions* are boolean expressions (the concrete syntax of the expression language can be defined by the modeller). Rules are comparable to constraints that ensure that certain conditions always hold.

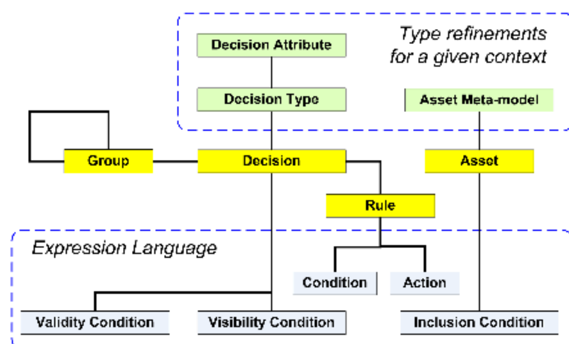


Figure 1. Constructs used in DoVML.

Product Line variability models built using DoVML are constructed such that they can be used for highly automated product derivation processes. The structure of the decision models and the concepts used therefor show high resemblance to process modelling approaches. As for example: *(i)* Visibility conditions are used to distinguish between decisions which are relevant for the user and the ones which are not. This guides the user through a product derivation process. *(ii)* Decision attributes like questions, descriptions and images are used to communicate decisions to the user. *(iii)* Rules are executed automatically to ensure the consistency of the decision making procedure.

2.1. The notion of a decision

A *decision* is a set of choices available at a certain point in time and arises whenever for a given goal there exist two or more ways of achieving it. Decisions can be used to represent the variation points in a product line model, and serve basically two purposes: *(i)* documenting and planning variability in the development phase and *(ii)* guiding users and automating product configuration during derivation phase. The process of taking a decision involves judging the merits of multiple options and selecting one of them for action (e.g., based on a consideration of customer requirements). In other terms a decision making process leads to the selection of a course of actions among several available alternatives.

Decisions are not independent of each other and cannot be made in isolation for two reasons: *(i)* Due to the dependencies surrounding a given decision, many decisions made earlier lead to new decisions and *(ii)* Many decisions are limited (constrained) depending on the context of already taken decisions.

In our modelling approach, we take care of two kinds of dependencies among decisions. Firstly, we need to be aware of the fact that not all decisions are equally important or relevant at a certain time. We therefore need constructs to model the hierarchy of decisions. Secondly, taking a certain decision may have implications on other decisions which also need to be considered (constraints). We therefore need to take care of the factors that influence the decision making process itself.

2.2. Decision vs. decision variable

For modelling purposes, we sometimes refer to a decision as a *decision variable*. A decision is a variable (like in programming languages) enriched with information regarding:

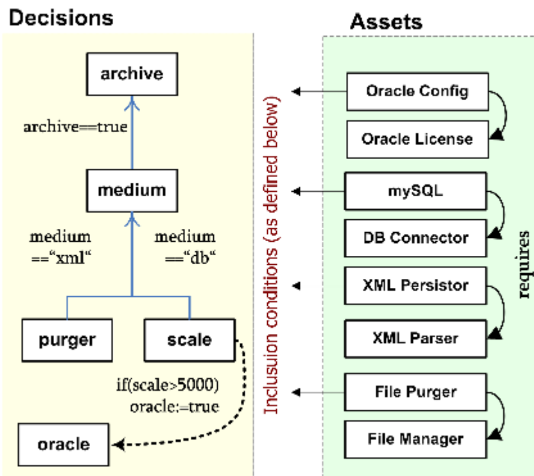
1. the set of possible values (including infinite sets, multiple ranges, and/or range constraints)

2. the specification of its position in the decision hierarchy (in relation to other available decisions)

3. the specification of the implications of taking the decision (on other decisions) and

4. labels and annotations (information for the user to better understand the decision).

Therefore, taking a decision is equivalent to binding a variable to a value.



Partial declaration of decisions (Problem Space)

Boolean decision **archive** visible if true;
Question: Do you want to archive daily records?

Enumeration decision **medium** visible if archive==true;
Question: Which medium should be used for archiving records?

Boolean decision **purger** visible if medium=="xml";
Question: Do you want old xml files to be automatically deleted?

Integer decision **scale** visible if medium=="db"; valid if scale>=0;
Question: Enter the anticipated number of daily records!
Post-script: if(scale>5000) oracle:=true;

Integer decision **oracle** visible if false;
// state variable, not visible to the user

Partial declaration of assets (Solution Space)

Component **XML Persistor** included if medium=="xml";
requires {XML Parser}

Component **mySQL** included if medium=="db" && scale<5000;
requires {DB Connector}

Component **File Purger** included if purger==true;
requires {File Manager}

Figure 2. Simplified representation of a variability model in DoVML.

2.3. The notion of an asset

Assets are used to describe the set of artefacts and their dependencies that are available in a certain development environment. The structure and organization of the solution space is specific to the domain/industrial context at hand, therefore the core of DoVML can be parameterized with an asset-meta model. Our approach doesn't assume fixed types of assets for modelling variability. By providing an abstract conceptual representation of structured data, the modeler defines the "modeling language" for the solution space. Due to lack of space in this paper, we omit the details about asset meta-modelling. In our modelling approach, the assets are linked to decisions via *inclusion conditions*, which are arbitrary boolean expressions built using the decision variables.

Figure 2 depicts a simplified representation of a variability model. It depicts the two key modelling elements (decisions and assets). The types of decisions in use (Boolean, Enumeration, Integer etc.) and the types of assets in use (Components, Resource, etc.) have been ommitted. For a better understanding of the terms and concepts presented in this paper, figure 2 is used as an running example. In the example, we assume a simple concrete syntax of the different expressions in use and the kind of relationships between different assets (e.g. requires) to be fixed.

2.4. Key concepts

In this paper the constructs of DoVML are defined using elementary set theory. We use the terms *types*, *variables* and *expressions* in the same way as in typed λ-calculus and functional programming languages. This means that expressions do not have side-effects and variables are bound to values. It also means that complex expressions are built from variables and simpler sub-expressions, by means of functions and operations. To give an abstract definition of decision models, it is not necessary to fix the concrete syntax in which the modeller writes the expressions, and thus we shall assume that such a syntax exists (together with well defined semantics). It is now possible in an unambiguous way to talk about the following:

1. The *type of a decision* υ is denoted by $\tau(\upsilon)$ and the *type of an expression* ε is denoted by $\tau(\varepsilon)$. For a type τ, we also use τ to denote the set of elements in τ.

2. The set of decisions involved in an expression ε is denoted by $\mathbb{V}(\varepsilon)$. This set of variables only includes the free variables, i.e., those which are not

bound internally in the expression (e.g. by local definition).

3. For a set of decision variables $\{\upsilon_1, \upsilon_2, .., \upsilon_n\}$, *the binding of the decisions* in the set is denoted by $\beta=\langle\upsilon_1{:}\eta_1,\ \upsilon_2{:}\eta_2,\ ..,\ \upsilon_n{:}\eta_n\rangle$. It is required, that $\eta_i\in\tau(\upsilon_i)$.

4. Furthermore $\mathbb{E}_B(\mathtt{S})$ is defined as the set of Boolean expressions (terms and formulae), that can be built using the variables in the set $\mathtt{S}$. In other words $\forall\varepsilon\in\mathbb{E}_B(\mathtt{S})$: $\tau(\varepsilon)\in\mathbb{B}\wedge\mathbb{V}(\varepsilon)\in\mathtt{S}$, where $\mathbb{B}=\{\mathtt{true},\mathtt{false}\}$.

3. Variability modelling with DoVML

DoVML needs to be parameterized (configured) to the specifics of the domain, before it can be used to model the variability. Such configurability of the language provides us with the flexibility required to adapt the approach to the needs of different variability implementation practices. We therefore define Σ, $\mathbb{L}$, $\mathbb{A}$ and AMM as needed, where

Σ is a finite *set of data types*, specifying the types of variables to be used in the model, e.g. Boolean, Enumeration, String, Double, Character etc. This set can be extended with other types, as required by the domain. E.g., more complex compound data types (e.g. Date and Time) are also possible. In the overview depicted in figure 1, the set Σ is represented by *decision types*.

$\mathbb{L}$ is a finite *set of labelling functions* providing detailed information for every decision variable. Such annotations have no formal meaning, but are helpful in understanding the model. Examples of such labels are- description of υ, images and URLs to elaborate the meaning of υ to the user, the question which the user is asked etc. Use of labelling functions (as compared to unstructured text-tags) helps in better interpretation of the tags. In the overview depicted in figure 1, such labels represent the *decision attributes*.

$\mathbb{A}$ is a finite *set of actions*, which are carried out upon taking decisions defined in the decision model. Read only actions are used to validate the actual status of the decisions taken by the user (e.g. assertions that can be made in order to make sure that certain constraints are fulfilled). Other actions can be used to make changes in the model: variables can be bound to new values and other properties of decisions can be changed. The execution semantics of the action should be provided with its definition. Actions can be compared to domain-specific functions for manipulation of decisions.

AMM is the meta-model of the assets, whose variability needs to be modelled. It is comparable to "entity-relationship models" in relational databases. The asset meta-model specifies (defines) the language for describing the solution space.

A decision model (DM) is a set of decision variables of the defined types $\forall\upsilon\in\mathtt{DM}:\tau(\upsilon)\in\Sigma$. Every decision variable in a decision model is a unique identifier and can be bound to a certain set of values. The names have no formal meaning but they have huge practical importance for the readability of a decision model (just like the use of mnemonic names in traditional programming). The range of possible values is partly specified by the type of the variable.

Furthermore the decisions in DM are specified in more detail using f_{val}, f_{vis} and $\Re$ where

1. f_{val} is a validity function restricting the range of variables $\forall\upsilon\in\mathtt{DM}:f_{val}(\upsilon)\rightarrow\mathbb{E}_B(\mathtt{DM})$.

2. f_{vis} is a visibility function specifying the hierarchy of decisions $\forall\upsilon\in\mathtt{DM}:f_{vis}(\upsilon)\rightarrow\mathbb{E}_B(\mathtt{DM}\backslash\{\upsilon\})$.

3. $\Re$ is a set of rules in the form "if condition then action" (e.g. $\mathbb{E}_B(\mathtt{DM})\Rightarrow\langle\upsilon_1:\eta_1\rangle$, where a condition implies a binding).

Using the asset meta-model AMM defined for the domain at hand, we also create an asset model AM, which describes the set of available artefacts. We associate a function f_{inc} to every asset α in the asset model, which specifies when α needs to be included in the final product $\forall\alpha\in\mathtt{AM}:f_{inc}(\alpha)\rightarrow\mathbb{E}_B(\mathtt{DM})$.

3.1. Validity condition $f_{val}(\upsilon)$

The set of possible values of a variable specified by the type of the variable is often too broad. As an example let us consider a decision υ, where $\tau(\upsilon)=\mathbb{R}$. In order to further restrict the range of the variable, one can make use of a validity condition, a Boolean expression involving variables in DM, $f_{val}(\upsilon)\rightarrow\mathbb{E}_B(\mathtt{DM})$. A validity condition $f_{val}(\upsilon)$ of a decision can be seen as the post-condition which has to be fulfilled after υ is bound to a certain value. Using validity conditions, it is possible to specify multiple ranges too (e.g. $f_{val}(\upsilon)\rightarrow(\upsilon\geq\eta_1\wedge\upsilon\leq\eta_2)\vee(\upsilon\geq\eta_3\wedge\upsilon\leq\eta_4)$). It can therefore also be seen as a range constraint, which is evaluated before a variable binding can take place. A binding $\beta=\langle\upsilon{:}\eta\rangle$ is valid if $\eta\in\tau(\upsilon)\wedge f_{val}(\upsilon)$, for $\forall\upsilon\in\mathtt{DM}$.

Example: For a decision variable υ, $\tau(\upsilon)=\mathbb{R}$ the validity condition could be defined as $f_{val}(\upsilon)\rightarrow(\upsilon\texttt{\%2=0})$, which would mean, that only even numbers are valid values of the variable. In figure 2, we make use of a validity condition for decision `scale`, $f_{val}(\texttt{scale})\rightarrow(\texttt{scale}\geq 1)$, meaning that only positive number from $\mathbb{Z}$ are valid values of `scale`.

3.2. Visibility condition $f_{vis}(\upsilon)$

For each decision variable $\upsilon\in\texttt{DM}$, there exists a visibility function $f_{vis}(\upsilon)$, which specifies, when a certain decision can be taken by the user at a certain point in time during derivation. The visibility condition needs to be evaluated, before a value has been assigned to the variable υ, so the expression returned by $f_{vis}(\upsilon)$ must not contain the variable υ itself. In other words, $\forall\upsilon\in\texttt{DM}$: $f_{vis}(\upsilon)\in\mathbb{E}_B(\texttt{DM})\wedge\mathbb{V}(\varepsilon)\subseteq(\texttt{DM}\backslash\{\upsilon\})$.

Hierarchy based on visibility conditions: The hierarchy of decisions (the order in which the decisions need to be taken) is partly specified by f_{vis}. To elaborate on the effects of visibility conditions, we define a relationship $\diamond$ between decision variables with respect to their visibility conditions. A variable υ_1 is said to have a $\diamond$ relationship to another variable υ_2, if the variable υ_2 appears in the visibility condition of υ_1. This kind of relationship between variables, which is written as $\upsilon_1\diamond\upsilon_2$ (read as υ_1's visibility depends on υ_2) is given if $\upsilon_2\in\mathbb{V}(f_{vis}(\upsilon_1))$. $\diamond$ is non-reflexive (the visibility condition of a variable cannot depend on itself), strictly anti-symmetric (variables cannot depend on each other) and transitive.

Example: In figure 2, the decisions are organized in a hierarchy based on their visibility conditions. Lets consider the decision regarding the medium to archive: $f_{vis}(\texttt{medium})\rightarrow\texttt{archive}$. The decision `medium` can be taken by the user only if the decision `archive` is bound to the value `true`. This implicitly requires, that the decision `archive` needs to be taken before `medium`. We can also note that $f_{vis}(\texttt{archive})\rightarrow\texttt{true}$ and $f_{vis}(\texttt{oracle})\rightarrow\texttt{false}$, which means these decisions are always/never visible to the user respectively.

3.3. State variables

Decisions which are never visible to the user, i.e. $f_{vis}(\upsilon)\rightarrow\texttt{false}$, are referred to as *state variables* and can be bound to their values only as a result of rules ($\mathfrak{R}$). Such decision variables can be used to keep track of different execution states of the model. They are bound to their values automatically as a result of executing the rules. Such rules help in aggregating values of decisions which have already been taken and allow to simplify complex expressions in models. For example (cf. figure 2) the decision variable `oracle` determining whether a oracle database is needed for the final system may be bound to a certain value automatically after the user decides on the size (`scale`) of the final system.

3.4. Specification of rules $\mathfrak{R}$

The effects of taking a decision (on other decisions) are modelled using a set of rules. Rules can be used basically for *(i) Assertion*, *(ii) Binding*, *(iii) Update* and *(iv) Information*. The semantic of rules used for assertion and binding is identical to constraints specified using boolean expressions in constraint satisfaction problems (CSPs). However, by using rules to update the model and to communicate to users at runtime, one can go beyond the borders of traditional constraints (as this is not the focus of constraints in CSPs). This also shows that variability models based on DoVML are created with the focus of an interactive product derivation process. The rules are specified in the form:

$$\texttt{if}\ \langle\texttt{condition}\rangle\ \texttt{then}\ \langle\texttt{action}\rangle,$$
where $\texttt{condition}\in\mathbb{E}_B(\texttt{DM})$ and $\texttt{action}\in\mathbb{A}$.

A rule is activated or triggered when its `condition` evaluates to `true`. Here we present a few examples of rules and their application. For the sake of simplicity in the examples, we assume the syntax of the rules to be similar to Pascal like programming languages. Rules could be used for:

(i) Assertion: Dependencies among decisions, where certain conditions always need to hold, e.g. a constraint in the form $(\upsilon_1=\eta_1)\Rightarrow(\upsilon_2=\eta_2)$ could be specified using the rule: $\texttt{if}(\upsilon_1=\eta_1)\texttt{then assert}(\upsilon_2=\eta_2)$ or simply $\texttt{assert}(\neg(\upsilon_1=\eta_1)\vee(\upsilon_2=\eta_2))$. The assert action is a read-only action. It does not change the value of the variables, but only makes sure that the condition holds.

(ii) Binding: Whenever there is a need to change the values of the variables we make use of binding actions. e.g. $\texttt{if}\ (\upsilon_1=\eta_1)\ \texttt{then setValue}(\upsilon_2,\eta_2)$. In general a binding action is comparable to a constraint as in CSPs, i.e. a condition implies a binding $\mathbb{E}_B(\texttt{DM})\Rightarrow(\beta=\langle\upsilon_1:\eta_1\rangle)$. In contrast to the assertion action, binding actions change the actual value of the decisions (i.e., they take decisions on behalf of the user). Here `setValue` is used as an example of a binding action (the actual syntax and semantics of all the actions is fixed when defining $\mathbb{A}$).

(iii) Update: Not only the values but also different attributes of decisions can be updated/manipulated using rules. As for example, depending on the

value of one decision, the validity condition of another decision might change, e.g. `if` $(\upsilon_1 = \eta_1)$ `then update(`$f_{val}(\upsilon_2) \rightarrow (\eta_2 \div 5 \neq 3)$`)`. Such an update action can be used to change the specification of model at runtime. Modification of the decision model itself as an implication of the decisions taken by the user can however also lead to problems regarding the determinability of the decision making procedure.

(iv) Information: Rules can also be used for informative purposes. By defining actions like `inform`, or `display` one can also capture knowledge which is required for the user during product derivation. Such rules have no formal semantics, but can be very helpful to the user to improve guidance during derivation. Example usage scenarios for this would be the creation of recommender systems based on variability models e.g. `if` $(\upsilon_1 = \eta_1)$ `then inform('It is recommended that ...')` [17].

3.5. Building an asset model

Asset models are instances of the asset-meta model describing the structure of the solution space. When building an asset meta-model the types of assets to be used, their attributes and dependencies among them can be defined. At this point, it is important to point out some peculiarities of the asset meta-models which we use in our approach.

Inclusion conditions: We associate a Boolean expression called inclusion condition to every asset α in the asset model `(AM)`. $\forall \alpha \in$ `AM`: $f_{inc}(\alpha) \rightarrow \mathbb{E}_B$(`DM`). Such an expression specifies the condition under which the asset α will be included in the final product. If an asset is always included in the system (e.g. utility classes, common libraries) then its inclusion condition is simply `true`. Considering the example presented in figure 2, the inclusion condition of the component `XMLPersistor` is defined as `medium==xml`. This means that the component `XMLPersistor` is included in the final product, if the decision `medium` is set to `xml`. The inclusion condition can be arbitrarily complex and can involve any number of variables, thus supporting not only 1:1 mappings between decisions and assets.

Basic dependency types: Often assets are not included or excluded from the final product directly because of decisions taken by the user but rather because of technical dependencies resulting from their implementation. For example (cf. figure 2), the component `FileManager` is included in the final product because it is required by the component `FilePurger`. In order to model such technical dependencies (functional and structural) we provide with a set of predefined relationship types (for automated interpretation of asset-dependencies). Examples of such basic relationship types are inclusion, exclusion, parent, child, predecessor, successor, implementation, abstraction etc. When specifying the asset meta-model for a certain organization, the modeler can define his own name for dependency types (e.g. "requires") and link it with a predefined type (e.g. "inclusion"). The naming of dependency types is similar to the concept of stereotypes in the UML.

When defining the semantics of decision-oriented variability models, we ignore $\mathbb{L}$, and f_{vis}, as they are primarily modeled with the focus of the product derivation process. We also don't care of the concrete syntax in which f_{val} and $\mathbb{A}$ are written. Further constructs like "roles of users", "configuration tasks" and project specific adaptations of the variability model [17] are out of scope of this paper.

4. Semantics of DoVML

A decision model represents the set of all possible `valid variable bindings` of the variables in the decision model, resolution of `DM` $\equiv \Upsilon$`(DM)`$=\{\beta_1, \beta_2, \ldots, \beta_n\}$, where n is possibly ∞ due to variables with infinite ranges. The process of taking decisions selects one possible binding from Υ. The resolution of a decision model is given by $\beta \in \Upsilon$.

A concrete binding $\beta \in \Upsilon$ can then be used for evaluating (calculating) the list of required assets. From the product derivation perspective, the assets can be seen as boolean variables, whose values are determined by the evaluation of their inclusion conditions. Every asset α can be interpreted as a boolean variable $\tau(\alpha) = \mathbb{B}$, whose binding is given by $\langle \alpha : f_{inc}(\alpha) \rangle$.

Furthermore, if asset dependencies are defined between assets α_1, α_2, α_3, such that α_1 `requires` α_2 `requires` α_3, then the dependency `requires` can be seen as a copy function, that assigns the same inclusion condition $f_{inc}(\alpha_1)$ to $f_{inc}(\alpha_2)$ and $f_{inc}(\alpha_3)$. i.e. α_1 `requires` α_2 `requires` $\alpha_3 \Rightarrow (f_{inc}(\alpha_3) \equiv f_{inc}(\alpha_2) \equiv f_{inc}(\alpha_1))$. The consequence is that if α_1 is included in the final product, then α_2 and α_3 are also included.

4.1. Interpreting/executing a variability model

The operational semantics of decision-oriented variability models can be explained by the algorithm in figure 3, which can interpret such variability models. The result of executing such a variability model is a set of taken decisions (binding of decision variables) and a set of assets required for the desired product.

Decision-making based on variability models (e.g.,

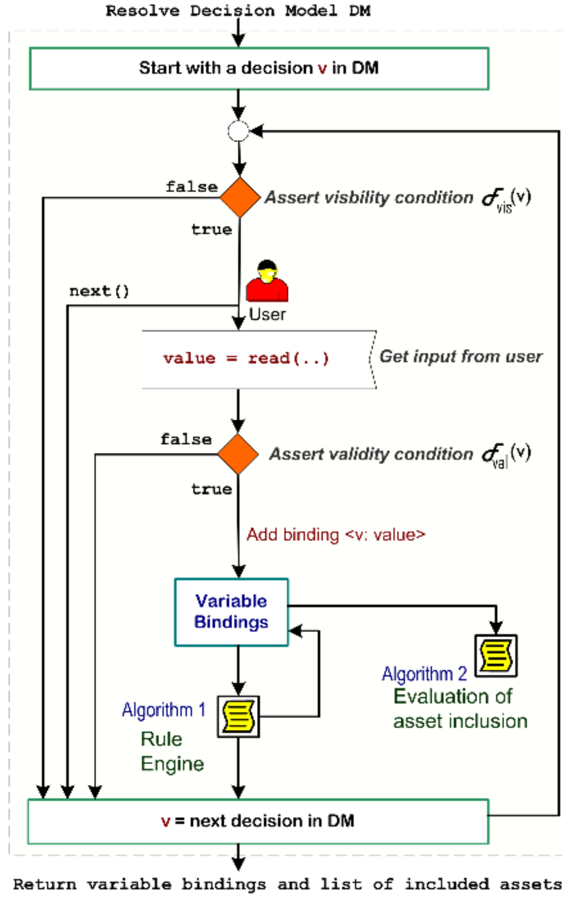


Figure 3. Overview of a sample algorithm for the execution of variability models.

as a part product derivation/configuration) is an interactive process. Decisions can either be visible or invisible to the user. The transition between these states is regulated by the evaluation of the visibility condition, which is triggered whenever a new variable binding takes place. All visible decisions are presented to the user. The variable binding takes place either as a result of user interaction or as a result of rules which are evaluated as required after a decision is taken. An asset can either be included in or excluded from the desired final product. The transition between these states occurs as a result of the evaluation of the inclusion condition of the assets.

Firstly, the visibility condition of each decision variable is evaluated. If the condition holds, then a question is presented to the user (possibly with other labels of the decision variable) so that the variable is better understood when taking the decision. The input from the user is evaluated against the validity condition. If the input was a valid one, then the variable is bound to the input value. Such a binding has two implications:

(i) *It triggers the rule engine*, which evaluates all the rules and executes them as necessary (overview of rule engine depicted in algorithm 1). Such rules can also cause a variable binding, which leads to a recursive call of the rule engine. So the execution of the action specified in the rule requires that the condition evaluates to `true`. The execution of the action can change the set of already bound variables; can however also only be informative. As the rule engine can trigger the evaluation of the rules again, it is important that there are no cyclic dependencies in the model. Cycles in the rules can be detected using standard cycle detection algorithms for graph like data structures.

(ii) *It triggers the evaluation of asset inclusion*, which is the process of figuring out which assets need to be included in the final product. The process (depicted in algorithm 2) consists of two phases: *(i)* evaluation of the inclusion condition and *(ii)* evalutation of asset dependencies. The set of included assets can then be used by domain-specific application generators simulators and deployment tools for further processing.

Algorithm 1 Sample evaluation of rules (rule engine)

Require: Binding $\beta \subseteq \{\upsilon_1{:}\eta_1, \upsilon_2{:}\eta_2, .., \upsilon_n{:}\eta_n\}$
  for all Rule ρ in $\Re$ **do**
    if ρ.condition holds **then**
      if ρ.action is of type `binding` **then**
        $\beta = \beta \cup \{\langle \rho\texttt{.action.}\upsilon{:}\rho\texttt{.action.}\eta \rangle\}$
        re-evaluate all rules
      else
        domain-specific interpretation of ρ.action
      end if
    end if
  end for
  return Binding β

Algorithm 2 Sample evaluation included assets

Require: Binding $\beta \subseteq \{\upsilon_1{:}\eta_1, \upsilon_2{:}\eta_2, \ldots, \upsilon_n{:}\eta_n\}$
  initialize set of included assets L
  for all Asset α in `AM` **do**
    if $f_{inc}(\alpha)$ holds $\wedge$ **then**
      $L = L \cup \{\alpha\}$
      {evaluate asset relationships}
      evaluate technical dependencies of α
    end if
  end for
  return set of included assets L

5. Implementing DoVML

The approach described in this paper has been implemented in a meta-tool for modelling variability called DecisionKing [7, 9]. In order to reflect on the current implementation status of the tool (from the perspective of the modelling language features), let us consider the overview diagram depicted in figure 1. In DecisionKing we have realised the abstract core elements of DoVML (i.e. Decisions, Assets, Groups and Rules) by providing simple implementations for exemplification.

Types of decisions (data types): DecisionKing currently supports four basic types of decisions– `Boolean` decisions are used to simulate yes/no questions. `Number` decisions are used mostly for parameter values, where the user decides on a numerical value. These are comparable to the type "double" in programming languages. Other numerical types: integer, short etc can be simulated using number decisions. `String` decisions are used for similar purposes as number decisions. They correspond to the data type "String" in programming languages. `Enumeration` decision can be seen as arrays of strings. Such decisions are used whenever different alternatives to the same variation point need to be modeled.

Decision attributes (labelling functions): Currently we support three decision attributes to communicate the meaning of a decision to the user `Descriptions` are blocks of text (e.g., in HTML) is used to clarify the meaning of a decison. HTML also allows the easy integration of images, videos and animations to improve guidance of product derivation process. `Questions` are formulated in a concise way in the user's problem space language, such that the answer to that question implies the value of the decision. By making use of `Annotations` one can attach arbitrary information (in textual form) to a decision.

Expression language (functions and actions): We are currently using an expression language, which shows high syntactic resemblance to Pascal. One can make use of standard operators (e.g. $+, -, \div, *, =, \neq, \leq, \geq, <, >, \vee, \wedge$, etc.) to build expressions. DecisionKing provides an expression editor (with syntax highlighting and auto completion, cf. figure 4) to ease the modelling process. Apart from the standard operators we provide the following actions to query the value of decisions and build more complex expressions. `setValue(d1, p1)` is an assignment function, which assigns the value p1 to decision d1. `selectOption(ld1, op1)`, `deselectOption(ld1, op1)` are used to select/deselect an alternative in a enumeration decision. `contains(ld1, op1)` is a set operator which can be used in enumeration decisions to simulate $\subset, \subseteq, \in$ operations. `allow(ld1, op1)`, `disallow(ld1, op1)` are used to expand/restrict the set of possible values in a enumeration decision. `isTaken(d1)` is used to query, whether a decision has already been taken by the user. `reset()` is used to retract a taken decision. Retracting a decision also resets all its implications modelled in the rules. These functions and actions add syntactic sugar to the actual implementation of the engine that evaluates the rules. We are currently using a rule engine based on JBOSS Rules[1]. All rules written using the functions defined above are translated into their corresponding representation in drools[2] notation. As for example the following rule

```
1 if (num_strands>4)then
2   setValue(casting_mode, {"Single"});
3 endif
```

is automatically translated into drools rule:

```
1  rule "0"
2  salience 0
3  no-loop true
4  when
5    num_strands:RuleNumberDecision(
6      name == "num_strands",
7      active==true)and
8      eval(num_strands.getPValue()>4)then
9        ArrayList<String> drools_a;
10       i.identify();
11       drools_a = new ArrayList<String>();
12       drools_a.add("Single");
13       i.set(0,"casting_mode",
14         new ArrayList<String>(drools_a));
15   end
```

Asset meta-models: As described earlier, for the description of the problem space, which is specific to implementation practices in different domains, our tool suite can be parameterized with a meta-model which is comparable to ER-models. Until now, we have created several such meta-models.

– *Siemens VAI*: For our industry partner we created a meta-model consisting of Components, Resources and Properties as asset types. Two dependency links (requires and contributes to) were used to describe the technical dependencies [7].

– *ERP System*: In order to model the variability of an enterprise resource planning (ERP) systems, we created a meta-model consisting of .NET "Plugins" as the basic asset type [21].

– *IAS System*: We also modeled the variability of an industrial automation system (IAS) to automate the runtime reconfiguration process [11].

– *DOPLER tool suite*: The variability of the DOPLER tool suite itself was modelled using DoVML. For this

[1] http://www.jboss.com/products/rules

[2] http://www.jboss.org/drools/

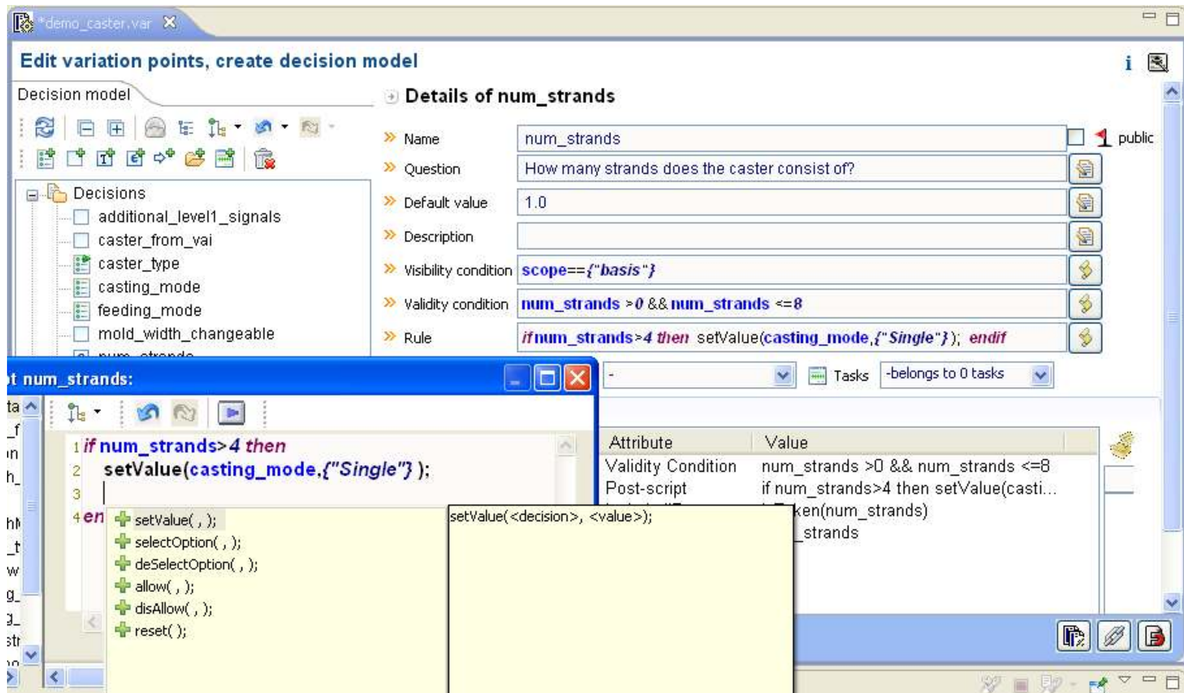


Figure 4. Modelling decision dependencies in DecisionKing.

purpose we create a meta-model consisting of Eclipse plugins, extension points and extension contributions as the asset types [12].

6. Summary and further work

In this paper we presented the details of our variability modelling approach based on decision modelling. Several publications in the past have already elaborated on the tools DecisionKing, ProjectKing and ConfigurationWizard, which are based on the modelling approach described in this paper.

Our modelling approach focuses on one of the primary goals of a product derivation process, i.e. the identification of the required assets to fulfil the needs of the customer specified in the form of taken decisions. This is however only one application area of variability models based on decisions. Flexibility and adaptability is introduced in our modelling approach by providing parameterization facilities for the language itself (e.g. by defining Σ, $\mathbb{L}$, $\mathbb{A}$ and $\mathtt{AMM}$ for each domain).

To illustrate the wide range of application areas, we have already used our approach and tools to automate the configuration of steel plant process automation software [7], to manage runtime adaptation of enterprise resource planning systems [21], to manage the lifecycle of industrial automation systems [11], and to monitor service-oriented systems at runtime [3].

We are currently working on more formal representations of decision-oriented variability models and their formal semantics. One longer term goal in this perspective is the formal definition and comparision to other available decision modelling approaches.

Apart from that, we are continuously extending the expression language used in our tool suite, which gives us the power to express variability constructs using functions at a higher level of abstraction. This includes implementation of different set operators, actions and other functions to model the dependencies among decisions/assets.

Ongoing work includes consistency checking and static analysis of decision-oriented models (e.g., by converting them into constraint satisfaction problems or petri nets) and further validation of the approach and tools in real world examples of our industry partner.

7. Acknowledgements

This work has been conducted in cooperation with Siemens VAI and has been supported by the Christian Doppler Forschungsgesellschaft, Austria. We would like to express our sincere gratitude to Stefan Wallner for his contribution in implementing the rule language based on JBOSS Rules.

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference (SPLC 2005)*, volume LNCS 3714, pages 7–20, Rennes, France, 2005.

[2] G. H. Campbell, S. R. Faulk, and D. M. Weiss. Introduction to synthesis. Technical report, Software Productivity Consortium, Herndon, VA, USA, 1990.

[3] R. Clotet, D. Dhungana, X. Franch, P. Grünbacher, L. López, J. Marco, and N. Seyff. Dealing with changes in service-oriented computing through integrated goal and variability modeling. In *Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)*, pages 43–52, Essen, Germany, 2008.

[4] S. P. Consortium. Synthesis guidebook. Technical report, SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium, 1991.

[5] K. Czarnecki, S. Helson, and U. Eisenecker. Staged configuration using feature models. In R. Nord, editor, *Lecture Notes in Computer Science, Software Product Lines, Third International Conference (SPLC 2004)*, volume LNCS 3154, pages 266–283. Springer-Verlag, 2004.

[6] D. Dhungana, P. Grünbacher, and R. Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *First International Workshop on Variability Modelling of Software-intensive Systems - Proceedings*, pages 119–128. Lero - Technical Report 2007-01, Limerick, Ireland, 2007.

[7] D. Dhungana, P. Grünbacher, and R. Rabiser. Domain-specific adaptations of product line variability modeling. In *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, 2007.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel. Dopler: An adaptable tool suite for product line engineering. In *11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, volume Second Volume, pages 151–152, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[9] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Tool Demonstration, 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, USA, 2007.

[10] T. Forster, D. Muthig, and D. Pech. Understanding decision models : Visualization and complexity reduction of software variability. In *Second International Workshop on Variability Modeling of Software-Intensive Systems*, volume 22, pages 111–119, 2008.

[11] R. Froschauer, D. Dhungana, and P. Gruenbacher. Managing the life-cycle of industrial automation systems with product line variability models. In *34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Parma, Italy, 2008.

[12] P. Grünbacher, R. Rabiser, and D. Dhungana. Product line tools are product lines too: Lessons learned from developing a tool suite. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, L'Aquila, Italy, 2008.

[13] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[14] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India, 2007.

[15] V. Myllärniemi, M. Raatikainen, and T. Männistö. Kumbang tools. In *11th International Software Product Line Conference (SPLC 2007), Tool Demonstration*, volume Second Volume, pages 135–136, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[16] R. Rabiser and D. Dhungana. Integrated support for product configuration and requirements engineering in product derivation. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07)*, Lübeck, Germany, 2007.

[17] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, 2007.

[18] K. Schmid and I. John. A customizable approach to full-life cycle variability management. *Journal of the Science of Computer Programming, Special Issue on Variability Management*, 53(3):259–284, 2004.

[19] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Minneapolis, MN, USA, 2006.

[20] D. Sellier and M. Mannion. Visualizing product line requirements selection decisions. In *11th International Software Product Line Conference (SPLC 2007), 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007)*, volume Second Volume, pages 109–118, Kyoto, Japan, 2007. Kindai Kagaku Sha Co. Ltd.

[21] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, February 2008. IEEE Computer Society.

Automated Analysis of Orthogonal Variability Models. A First Step

Fabricia Roos-Frantz [*†]
Universidade Regional do Noroeste do Estado do RS (UNIJUI)
São Francisco, 501.
Iju 98700-000 RS (Brazil)
frfrantz@unijui.edu.br

Sergio Segura
Department of Computer Languages and Systems
University of Seville
Av. de la Reina Mercedes S/N,
41012 Seville, Spain
sergiosegura@us.es

Abstract

The automated analysis of variability models is a challenge to be reached in SPLE (Software Product Line Engineering). Only recently researchers have devoted their attention to the reasoning on these models. However, their work has focused on Feature Models. Orthogonal Variability Modeling (OVM) is one of the approaches for modeling variability in software product line. Hence, an automated support is needed to reasoning on orthogonal variability models (OVMs). Although the automated analysis of OVMs has been proposed, it only deals with a small number of analysis operations, which are implemented using a specific logical representation and solver. In this position paper, we present the proposal that we will carry out to achieve an adequate tool to the analysis on OVMs. As part of this paper, we informally define some analysis operations on OVMs. In addition, we propose to study the possibility of extending FAMA framework for supporting analysis on OVMs. We consider that FAMA (FeAture Model Analyzer) could be a suitable option to automate this analysis since it provides a formal basis, integrate multiple solvers and already provide tools.

1. Introduction and Preliminaries

The automated analysis of variability models is a challenge to be reached in SPLE (Software Product Line Engineering). Although there are several kinds of variability models, the majority of the research works on analysis of these models has focused on Feature Models. In the literature, there are different proposals providing automated support for the analysis of feature models [3, 4, 8, 9, 10, 11, 13, 18, 22]. Each one of them use different logical paradigm or formalism to provide the automated support (e.g. description logic, propositional logic, constraint programming). Most of them use SAT, BDD or CSP off-the-shelf solvers to automate various analysis operations, e.g. , checking if a product is valid, checking if a model is void, detecting dead features, etc.

To the best of our knowledge only Metzger et al. [13] provide a automated support for the analysis of OVM. They introduce a formalization of OVMs and propose using SAT[1] solvers to automate analysis operations on OVMs. This proposal only deal with five operations and whose semantics is based on feature models. Besides, they use just one type of solver to automate the analysis. The various solvers (SAT, BDD and CSP solvers) have varying degrees of performance and coverage with regard to analysis operations [2, 5].

Benavides [4] proposes a formal framework FAMA-F for the automated analysis of software product lines in general and feature models in particular. The framework is independent of the variability model (VM) used for the analysis. FAMA-F integrate some of the most commonly used logic representations and solvers proposed in the literature (BDD[2], SAT[3] and CSP[4] solvers are implemented). It integrates different solvers in order to combine the best of all of them in terms of performance. We wonder if is possible to extend this framework to automate various reasoning tasks on OVMs, e.g., verifying if a product is valid, checking if a model is void, detecting "dead" nodes, etc. (see Sect. 3). We consider that FAMA-F could be a suitable option to automate the analysis of OVM since it provides a formal basis, integrate multiple solvers and has available tools.

To achieve a suitable automated support for the analysis of OVMs, we identify three issues to be addressed, namely: *i)* specification of operations, *ii)* formal representation of

*PhD student at the University of Sevilla
†Supported by the Evangelischer Entwicklungsdienst e.V. (EED)

[1] They use SAT4j solver http://www.sat4j.org
[2] JavaBDD solver, http://javabdd.sourceforge.net
[3] SAT4j solver http://www.sat4j.org
[4] Constraint Satisfaction Problem www.4c.ucc.ie/

model and operations, and *iii)* implementation of operations. These aspects motivated some of the main research questions to be addresses in our future work.

1.1. OVM (Orthogonal Variability Model)

OVM is a proposal for documenting software product line variability [14]. In an OVM only the variability of the product line is documented. In this model a *variation point (VP)* documents a variable item and a *variant (V)* documents the possible instances of a variable item. All VPs are related to at least one V and each V is related to one VP. Both VPs and Vs can be either optional or mandatory (see Figure 1). A mandatory VP must always be bound, i.e, all the product of the product line must have this VP and its Vs must always be chosen. An optional VP does not have to be bound, it may be chosen to a specific product. Always that a VP, mandatory or optional, is bound, its mandatory Vs must be chosen and its optional Vs can, but do not have to be chosen. In OVM, optional variants may be grouped in *alternative choices*. This group is associated to a cardinality *[min...max]* (see Figure 1). Cardinality determines how many Vs may be chosen in an alternative choice, at least *min* and at most *max* Vs of the group. Figure 1 depicts the graphical notation for OVMs [14, 13].

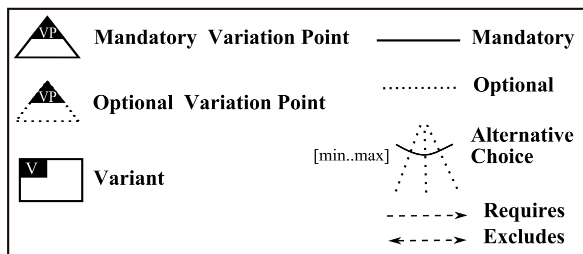


Figure 1. Graphical notation for OVM

In OVM, constraints between nodes are defined graphically. A constrain may be defined between Vs, VPs and Vs and VPs and may be an *excludes* constraint or a *requires* constraint. The excludes constraint specifies a mutual exclusion, for instance, a variant *excludes* a optional VP means that if the variant is chosen to a specific product the VP must not be bound, and vice versa. A *requires* constraint specifies an implication, for instance, a variant *requires* a optional VP means that always the variant is part of a product, the optional VP must be also part of that product. Figure 2 depicts a example of an OVM inspired by the mobile phone industry.

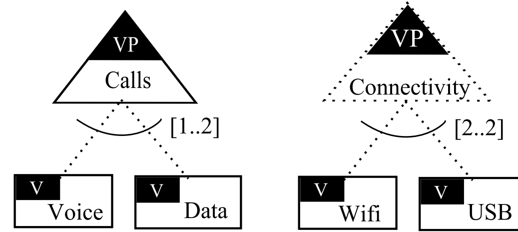


Figure 2. OVM example: mobile phone product line

1.2. Automated Analysis of OVM

To the best of our knowledge, there is only one proposal dealing with the automated analysis of OVM [13]. Metzger et al. are working in a tool support for variability management, which offers support for the analysis of OVM. Their prototype uses the off-the-shelf SAT solver library SAT4J. This SAT solver request a Boolean formula in CNF (conjunctive normal form) and delivers all variable assignment that evaluate the input formula true. If no such assignment exists, the formula is unsatisfiable. This proposal provides analysing of only five operations and using one solver. Furthermore, it makes the automated reasoning on OVM using the VFD semantics.

VFD (Varied Feature Diagram) is based on FFD (Free Feature Diagrams) which is a parametric construct designed to define the syntax and semantics of FODA-inspired FD (Feature Diagram) languages in a generic way [16, 17]. Metzger et al. propose reusing this formalization of feature diagrams, in other words VFD, to introduce a formalization of OVMs. They introduce a formal version of OVMs abstract syntax and describe a translation from OVM to VFD, thereby they give OVM a formal semantic.

1.3. FAMA framework

FeAture Model Analyzer (FAMA-F), proposed by Benavides [4], is a formal framework for the automated analysis of software product lines in general and feature models in particular, in other words, this is a framework independent of the variability model. This framework defines different reasoning operations on feature models, like calculating the number of products in a Software Product Line (SPL), getting a list of its products, filtering products according to a criterion or detecting and explaining errors. Thus, we presume it could be extended with OVM. It is defined with a high abstraction level, provides support for the most extended feature model notations and can be extended with new operations and solvers as needed.

The FAMA-F is defined in four layers from a higher (i.e. abstract foundation layer) to a lower abstraction level (i.e. implementation layer), see Figure 3.

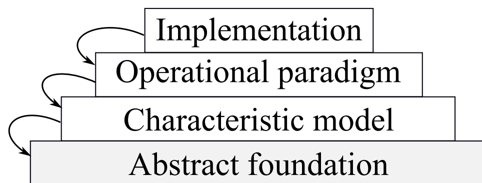


Figure 3. The four-layers FAMA-F

The FAMA-F abstract foundation layer provides an abstract and formal definition of software product lines and the operations of analysis that can be performed on them. The abstract foundation layer defines characteristic models as those that describe the allowed products configurations of the software product line. In the FAMA-F characteristic model, a specific variability model must be formally defined. Up to now feature models are the only variability model considered. In this layer the semantics of a specific feature model (FAMA-FM) is formalized with a formal language Z. The operational paradigm layer depends on the variability model used and provides a logical representation to the semantics of feature model and analysis operations, it allows using different logical representation. In the implementation layer, a translation from the logical representation described in the operational paradigm layer to the real solvers is provided. FAMA-F allows the usage of multi solvers to resolve a logical representation, may be used CSP, SAT and BDD solvers.

Additionally, Benavides et al. [7] presented an implementation of this framework, the *FAMA Eclipse plug-in* (FAMA-EP)[5]. It is an extensible tool for the automated analysis of feature models that integrated three logic paradigms and their respective solvers: Constraint Solver Programming (CSP) by means of JaCoP, Propositional Satisfiability (SAT) by means of SAT4j and Binary Decision Diagram (BDD) by means of JavaBDD. FAMA-EP allows the integration of different logic representations and solvers in order to optimize the analysis process.

In addition to FAMA-EP, Benavides et al. [20] provide the *FAMA Framework (FAMA-FW)* with the intention of allowing third parties to integrate their automated reasoning techniques into a workspace where some basic features are provided by default. FAMA-FW is a tool for the automated analysis of variability models (VM). Its main objective is providing an extensible framework where current research on VM automated analysis might be developed and easily integrated into a final product. It is built following the SPL paradigm supporting different variability metamodels, reasoners or solvers, analysis questions and reasoner selectors, easing the production of customized VM analysis tools. FAMA-FW is the result of research presented in [8, 5, 2, 19].

[5]http://www.isa.us.es/fama

1.4. Structure of this paper

The remainder of this paper is structured as follows: Section 2 provides an overview about semantics OVM. Section 3 provides an informal specifications of some operations on OVM. In Section 4 we propose the use of FAMA framework as tooling to automated analysis of OVMs. Finally, we summarize our future research in Section 5.

2. What specific formal semantics of OVM should be used?

To carry out the automated reasoning on OVM, we need a well defined semantics of these model. There are several options to give formal semantics to OVM. One of them is translating OVM to a feature model, since the semantic of these has already been defined [16, 17]. Another way is using a formal language, such as Z [21] or B [1].

To the best of our knowledge, there is only one formal semantics OVM in the literature, proposed by Metzger et al.[13], which has been obtained through a translation from OVM to VFD. This way to give semantics to OVM makes its semantics dependent of VFD. Such dependency may result in an drawback, due to the increase in the model size. When a OVM model is translated into a VFD model, a larger number of non-primitive nodes are generated and consequently the model becomes bigger, therefore it may damage the performance of the operations. Besides, one of the advantages that OVM offers is a significant reduction in the model size and complexity, because only the variable aspects of a product line are documented in a first-class model [15]. We wonder using VFD to give semantics to OVM contradict the advantage of OVM concerning the model size.

We will study other ways to give semantics to OVM aiming to evaluate these alternatives and have more conclusions. The first work will be on translating OVM to the feature model metamodel used in FAMA-F. This approach would allow us to reuse the formal foundation and support multi solver [6] provided by FAMA-F for reasoning on OVM. The other alternative is to study the usage of a formal specification language Z or B, without the translation to another language.

3. Operations on OVM

In the last years, different analysis operations over feature models have been identified [5, 4, 17, 3]. We select some of them and informally describe these on OVM. These operations observe the properties of a model without modifying it, they take a OVM as an input and provide a response as a result. Next, we define informally those operations, namely:

Valid product. This operations checks whether a given product belongs to the set of products represented by the OVM or not. For instance, let us consider the products P1 and P2, described below, and the OVM of Figure 2.

P1 = {*Connectivity, USB, Wifi*}

P2 = {*Calls, Data, Connectivity, USB, Wifi*}

The product P1 is not a valid product for the OVM since it does not include the mandatory variation point *Calls*. On the other hand, the product P2 is a valid product for the OVM because it is included in the set of products represented by the model.

Void OVM. This operation checks whether a OVM is void or not, i.e. if it represents at least one product. The reasons that may make a OVM to be void are related with a wrong usage of the constraint dependencies. As an example, Figure 4 depicts a void OVM. The excludes_VP_VP constraint makes not possible the selection of the mandatory VP *Functions*, what adds a contradiction to the model.

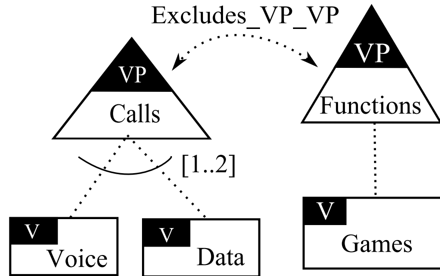


Figure 4. Void OVM

Core nodes. This operation returns the set of nodes (variants or variation points) that appear in all products of the software product line. For instance, the set of core nodes of the OVM presented in Figure 2 is {*Calls*}.

Dead nodes. This operation returns a set of dead nodes (if any), i.e. those that do not appear in any product. Dead nodes are caused by a wrong usage of constraint dependencies and are the responsible of making a OVM to be void. The Figure 5 depicts some common cases of dead nodes on OVMs. Dead nodes in the figure are labeled with D.

All products. This operation returns all the products represented by a model. As an example, the set of all the products of the OVM presented in Figure 2 is detailed below:

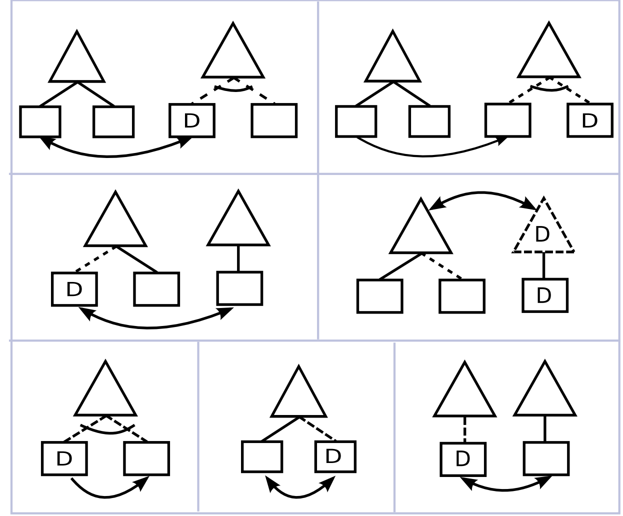


Figure 5. Common cases of dead nodes on OVM

P1 = {*Calls, Voice*}

P2 = {*Calls, Voice, Connectivity, Wifi, USB*}

P3 = {*Calls, Voice, Data, Connectivity, Wifi, USB*}

P4 = {*Calls, Data*}

P5 = {*Calls, Data, Connectivity, Wifi, USB*}

To automate the computation of those operations identified we have to formally define them. The formalization found in the literature for this operations were defined according to VFD semantics [13]. Metzger et al. suggest formal definition to the operations: *void model*, *valid product*, *core nodes*, *dead nodes* and *all products*. We will study how to formalize these and all the others operations identified in relation to OVM.

3.1. How to specify the equivalent models operation on OVM?

The equivalent models operation checks whether two models are equivalent. Two models are equivalent if they represent the same set of products [4]. If we observe the example depicted in the Figure 6, we can say that both models are equivalent, because they represent the same set of products. We believe that this operation is not correct to OVM because a *variant* is different of a *variation point*. In the product of the first model, *Media* is a *variation point* and in the second one, *Media* is a *variant*. Therefore, we believe that these models represent different set of products, then the equivalent operation should be redefined for OVM.

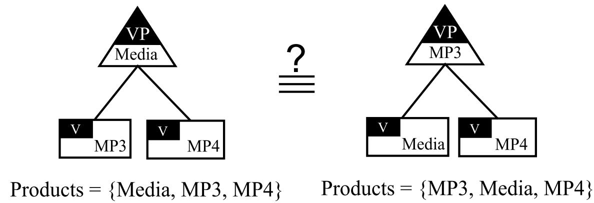


Figure 6. Equivalent models?

3.2. How to specify the merging operation on OVM?

We conclude that a new concept of merging operation can be applied to OVM, the merging into parts. This operation takes as input one part from one OVM model and another part of another OVM model and returns a new OVM model with the merging of those parts.

Three kinds of merging operations on feature models were identified by Schobbens et al. [17], particularly: Intersection, Union and Reduced product. The former, returns a feature model that encompasses the products included in both inputs models. The second, returns a feature model that encompasses all the products included in any of the inputs models. The later, returns a feature model including all the products of the input models plus all the new possible feature combinations. The merging of models may be helpful in a collaborative environment in which different people modify the model concurrently [12].

The Figure 7 depicts a visual example of the merging into parts operation on OVM, which is based on the merging operations proposed in [17]. According to the example, the *Connectivity VP* of the model *A* allows configuring three products: P1{Connectivity, Wifi}, P2{Connectivity, USB}, and P3{Connectivity, Wifi, USB}. The *Connectivity VP* of the model *B* also allows configuring three others products: P1{Connectivity, Wifi}, P2{Connectivity, Bluetooth}, and P3{Connectivity, Wifi, Bluetooth}. Then, if we merge both *Connectivity VPs*, we will have three new models as the result of the merging operation.

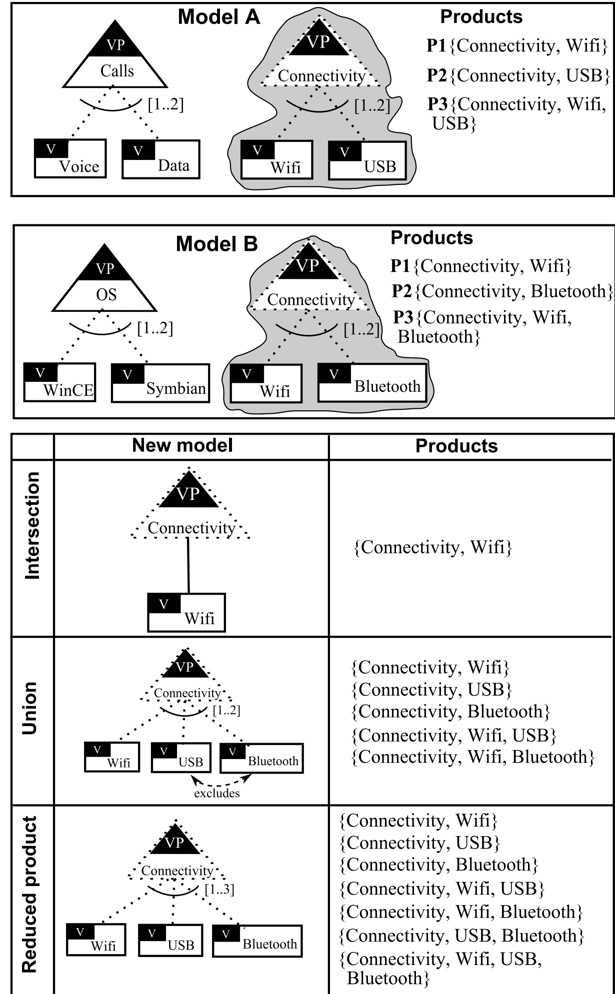


Figure 7. Merging into parts on OVM

4. Tooling

We will study how to extend FAMA-F [4] to support the analysis of OVMs. FAMA-F is a framework supporting the usage of different logics paradigms and solvers in order to optimize the performance of the analysis process. This framework is independent of the type of variability model, e.g. it is possible to use feature models or OVM. The first step to be done is to define the FAMA's characteristic model layer using OVM as variability model of SPL. At this layer OVM must be formally defined using the specification language Z. Afterwards, OVM must be translated by a logical representation. In particular, in the operational paradigm layer of FAMA-F, a OVM will be translated into a generic Constraint Satisfaction Problem (CSP). At last, we have to translate from the abstract CSP to the real CSP solver. At this framework multiples solver can be used: CSP, SAT and BDD solvers. Additionally, we can use the tool FAMA-FW as a support for implementing our tooling.

5. Future work

We will study a way to achieve a suitable tooling to analysis of OVMs. To do this, we intend to study what is the adequate semantics OVM to be used. Besides, we will specify all the operations that may be applied to OVM and to formally define them. Additionally, as one of the main moti-

vations of our research, we will study a possible extension of the FAMA framework for supporting analysis operations on OVM.

6. Acknowledgement

This work was partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and by the Andalusian Government under project ISABEL (TIC-2533).

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005.

[4] D. Benavides. *On the automated analysis of software product lines using feature models*. PhD thesis, University of Sevilla, 2007.

[5] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 367–376, 2006.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms (SPLC'06)*, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[8] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.

[9] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Lines Conference (SPLC 2007)*, volume 0, pages 23–34. IEEE Computer Society, 2007.

[10] S. Fan and N. Zhang. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 4252, pages 1144–1151. Springer, 2006.

[11] M. Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer.

[12] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[13] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007.

[14] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Fundations, Principles and Techniques*. Springer–Verlag, Berlin, DE, 2005.

[15] K. Pohl, F. van der Linden, and A. Metzger. Software product line variability management. In *SPLC*, page 219, 2006.

[16] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006. IEEE Computer Society.

[17] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2007.

[18] J. Sun, H. Zhang, Y.-F. Li, and H. H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the ICECSS05*, pages 303–312, 2005.

[19] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[20] P. Trinidad, D. Benavides, A. Ruiz-Corts, S. Segura, and A. Jimenez. Fama framework. In *Software Product Line Conference Tool Demonstrations (SPLC 08 Tools Demos) (in press)*, 2008.

[21] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

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

A Method to Analyze Variability Based on Product Release History: Case Study of Automotive System

Kentaro Yoshimura, Fumio Narisawa, and Koji Hashimoto
Hitachi Research Laboratory, Hitachi, Ltd.
(MD#244) 7-1-1 Omika, Hitachi, Ibaraki 319-1292, Japan
kentaro.yoshimura.jr@hitachi.com

Tohru Kikuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
kikuno@ist.osaka-u.ac.jp

Abstract

Variability is a keystone for developing reusable software product line (SPL) assets. Top-down approaches like feature-oriented analysis are widely used for detecting variability. However, when we introduce the SPL approach into existing products, analyzing variabilities of requirements and comparing them to legacy assets depend on the expertise of the analysts and are time consuming.

We present an complemental approach to analyze the variability candidates from existing product release history. We apply factor analysis method to detect co-change patterns of the legacy artifacts across product releases, and to suggest the variability candidates. To examine the applicability of our approach, we conducted an experimental application using a software repository of automotive engine-control software. As a result of the experiment, four variabilities and their variation points are detected successfully.

1 Introduction

SPL approach has been proposed as a development method for software that has variations. For reusing software across a product line, variabilities that are differences in requirements between product variations are analyzed. For migrating from existing products to SPL, understanding variability of the existing products and variation points of the artifact is an important practice.

Research in SPL engineering has mostly focused on the construction of product line infrastructures and activities based on requirements of future products: scoping, domain analysis, architecture creation, and variability management [2][3][8]. On the other hand, existing products contain a lot of domain expertise and are reliable from an industry point of view. The commonality and variability analysis for existing software is one of the most important issues to define a future product line while reusing existing software.

Variability analysis of existing software is a block against migrating into SPL engineering. Related studies have proposed methods for analyzing commonality and variability from a requirement point of view and connecting that to the implementation [6][7]. However, requirements and implementations of existing software are enormous. Moreover, such requirement analysis strongly depends on the expertise of the analysts and is time consuming. Some industrial case studies reported that commonality and variability analysis takes a long time[11][12]. There is a need for analyzing the commonality and variability in an automatic way.

We focus on the release history of existing products. Existing products contain their variability in their change history. If we could extract the variability from the existing products, that will be useful information for migrating the software product line because most of the existing variability information will also be valid in future product lines. Specifically, we analyze the product release history. When the bindings of variability were different between products releases, the realization of the software components that relates to the variability was also different. A major variability should occur several times, and the relationship to the software components (variation point) could be detected as a significant change pattern in the history.

We developed a method to suggest variability candidates across existing software with analyzing the co-change

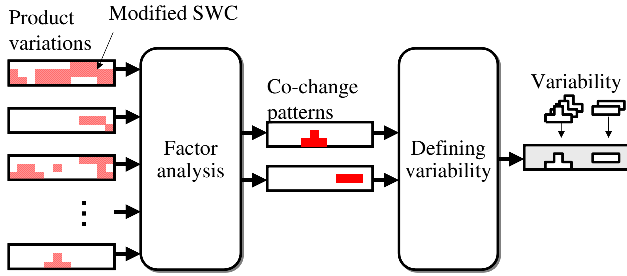


Figure 1. Overview of proposed method

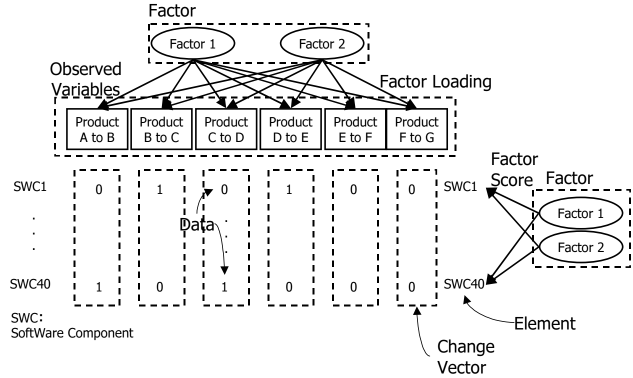


Figure 2. Overview of factor analysis

pattern[14]. The idea is inspired by factor analysis. Factor analysis is a multivariable analysis technique used to explain commonality among observed variables in terms of fewer unobserved variables called "factors." The observed variables are modeled as linear combinations of the factors plus "specific variables." The factor analysis extracts the co-change pattern as the commonality of changes between products and a set of software components that have changed with variability candidate. We thus tried to apply the factor-analysis technique to detect variability in existing products and call our approach "FAVE: Factor Analysis based Variability Extraction".

In this paper, we apply FAVE method to an industrial case study and discuss its results. The previous paper [14] describes FAVE method and applies it to an industrial, but small case study. In contrast, we apply the method to relatively large example in order to evaluate the method and to analysis the future works.

An overview of the proposed method and scope of this paper are shown in Figure 1. Inputs of FAVE are the change history of the products, and the output is variability in the product line. The factor analysis extracts the change pattern as commonality of changes and a set of software components that have changed with variability. After that, the detected variability is refactored as a variation point and its variants.

This paper describes a brief overview of the FAVE method and an experimental application to the software repository of automotive engine-control software. This paper is structured as follows: Section 3 summarizes related work. Section 2 describes a brief overview of the factor-analysis technique. Section 4 describes our FAVE method for detecting variability in existing products. The application of the proposed method to existing automotive control software is the topic of Section 5. Section 6 describes our work in progress and a conclusion.

2 Factor Analysis

An overview of the factor analysis is shown in Figure 2. Factor analysis is a well-known multivariable analysis technique used to explain commonality among observed variables in terms of fewer unobserved variables called "factors".

A factor loading is the degree of correlation between the observed variable and the factor. The factor has a strong relationship with the observed variable if its factor loading is significantly high.

Each factor also correlates with elements of the observed variables. The degree of correlation between the element and the factor is called a factor score. Using the factor score, we can expect that the meaning of the elements is based on the meaning of the factor.

3 Related Work

SPL engineering is an investment in future products, so various researchers have studied roadmap-oriented product line engineering[2][3][10]. There are also many studies about product line engineering for legacy software. Kang[7] analyzed the requirements of the legacy system and modeled them as a feature tree to migrate into SPL. John[6] analyzed documents of the legacy system to detect variability. Steger[11] analyzed electric and electronic architecture of the controller as variation points of the system. However, researchers have started mainly from current products to future products. In contrast, our approach focuses on the release history of the product line from the past to the present and analyzes how the product have varied. This result of the extraction helps developers understand the variability of the existing products.

We have already developed a method for evaluating commonality between two current products in a quantitative way [13]. In the proposed method, the commonality between products was detected, but we were unable to detect the variability across the variants. The difference between products consists of variability and product-specific parts. We could not separate them even when analyzing only two

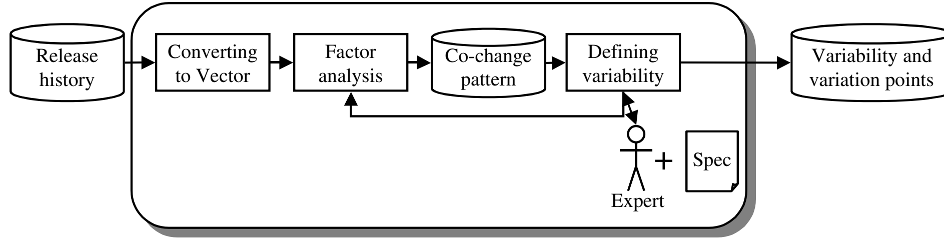


Figure 3. Overview of FAVE process

products. In contrast to our previous work, FAVE extracts the variability from the differences between multiple existing products.

In [4], Fischer proposed an approach to detect coupling between modules of multiple product variants by mining their code repositories. Their approach detects the degree of coupling between modules across the change history of the variants. In contrast to their work, FAVE detects the variability among the product release history in the orthogonal region and the variation points as correlation to the software components.

Loesch[9] presented a method to optimize variability provided in a product line. Their approach analyzes the usage of variable features in actual products derived from the product line. They applied a formal concept-analysis method for optimization. In contrast, our approach analyzes the release history of the existing products that have not been migrated into SPL yet.

Independently from SPL, Zimmermann[15] developed an approach to detect couplings of software components from the version history. However, their focus was on checking software components that were often modified at the same time. In contrast, our approach also clusters the change pattern of legacy software and extracts that as a variation point in the concept of SPL.

4 FAVE - Factor Analysis based Variability Extraction

4.1 Overview

The purpose of FAVE is to detect product-line variability that has already occurred in existing products. An overview of the FAVE process is shown in Figure 3. We have already developed the method in our previous work [14], so we describe the process briefly in this section.

Variability analysis is the first step for migrating existing products into software product lines. In this step, we analyze the existing product artifacts and classify them into common parts, variable parts, and product-specific parts. The aim of FAVE is the variability mining of existing products.

As shown in Figure 3, we convert product release histories to change vectors so that we could analyze them numerically. Then, we apply the factor analysis to the change vectors and identify co-change patterns as the variability candidates of the product line. After that, we define the means of the variability from the viewpoint of requirements and specifications.

In the following subsections, we explain briefly how FAVE is applied to existing products.

4.2 From Product Release History to Change Vectors

Product release history is not numerical data, so we cannot apply the factor analysis directly. For analyzing variability based on factor analysis, we convert the release history to change vectors.

An overview of the product release history is shown in Figure 4. Typically, an organization develop new product from the released product that has similiar requirements. Based on the difference of the requirements, e.g. different mechanical parts or new functionality, some software components (SWCs) are altered, added or deleted.

We define a change vector as a set of binary data that indicates difference in software components between product releases. For example, in Figure 4, software component 1, 2, 3 and 5 are changed from product A to B. Therefore, the change vector of product A to B is $(1, 1, 1, 0, 1, ...)^T$.

Product release history often branches because of the pararell development of product variations. In that case, change vector is generated from the difference between the last and the first product release over the branch point. For example, in Figure 4, software component 1, 4 and 5 are changed from product B to D. Therefore, the change vector of product B to D is $(1, 0, 0, 1, 1, ...)^T$.

After change vectors are generated based on changes between all serial product release, the vectors are used as observed variables for the factor analysis in the next step.

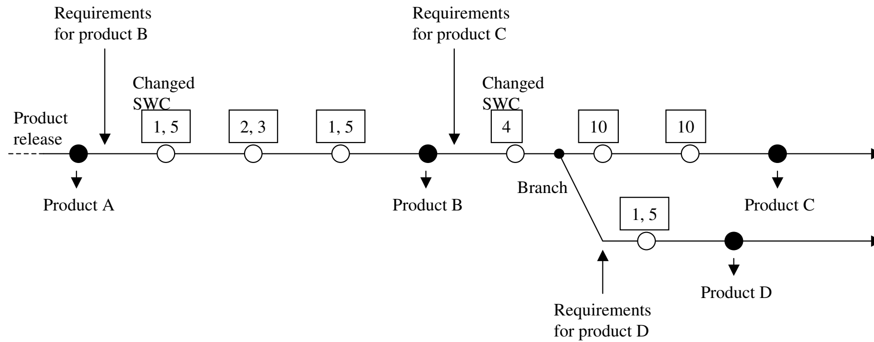


Figure 4. Product release hisotry

4.3 Factor Analysis

After converting the release history, we apply the factor analysis. As a result of the factor analysis, we obtain the following data.

- **Factor**
  A factor indicates a commonality of the changes between existing products. We can extract the factor as variability candidate in the product line because commonality of the change vector is a co-change pattern of how the existing software changes across the release history.

- **Factor Loading**
  A factor loading indicates which change vector is correlated with the variability candidate. When the factor loading is high, the variability candidate may be related to some specifications that changes in the change vectors.

- **Factor Score**
  A factor score indicates which software components are correlated with the variability candidate. Components that have high factor scores were modified together, and the timing is related to the variability candidate. This suggests that a cluster that consists of software components is a variation point of the variability candidate.

4.4 Defining Variability

A process of how domain experts define the means of the extracted variability based on the result of the factor analysis is shown in Figure 5.

First, significant factors are selected for defining the variability. The details of the selection process depend on the analysis method of the factor analysis such as the promax method and varimax method. For example, the number of significant factors is decided by using the chi-squared test and cumulating ratio of the factors.

Next, we analyze the product release histories that correlate the factors. We select the change vectors that have high factor loadings. The factor may be a variability that occurred in the selected change vectors between the existing products. In this step, we do not go into the details of the release, but define which change vectors are related to the variability.

Then, we select the set of software components that indicates the variation point. If the factor score is significantly high, the software component has been modified when a variability occurred in release histories that has a high factor loading. In the context of SPL, a set of the software components means the variation point of the variability. In the next step, we also speculate on the meaning of the variability candidate by referring to the combination of the software components' features.

As the last step, the candidate is interpreted as a variability of the SPL. For example, we guess the meanings of the candidate as indicating the abstracted feature from the set of the software components that have high factor scores. Then, we check whether the variability has occurred in the change vectors that have a high factor loading. Although only this step is an iterative step and depends on the expertise of an analyst, the result of the factor analysis will be helpful information for the analyst.

5 Case Study

5.1 Overview

An overview of an engine-control system is shown in Figure 6. The system monitors engine status and driver requests, and controls the engine by regulating the amount of fuel injection, ignition timing, and quantities of intake air, for example. There are a number of variations that correlate to the mechanical structure, which satisfy the product spec-

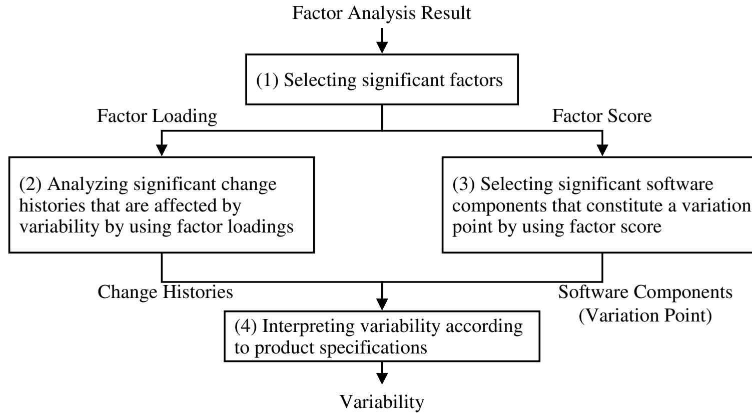


Figure 5. Process for defining variability

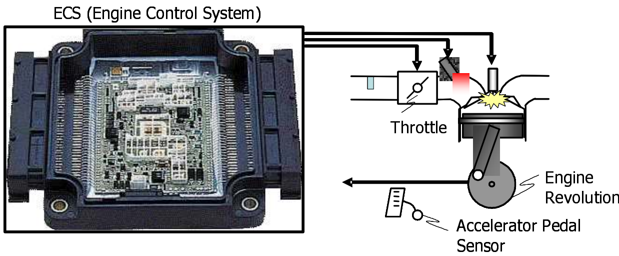


Figure 6. Engine-control system

Table 1. Case study example

Application	Engine-control software
# of products	16
# of software components	49

Table 2. Case study environment

Environment	R ver 2.6.1
Method	Maximum likelihood estimation
Rotation	Orthogonal (varimax)
Scores	Regression

ification, e.g., number of cylinders, transmission type, fuel type, and fuel injection.

We apply FAVE to part of the engine-control application layer. Generally, an engine-control system consists of a basic software layer and application software layer. As defined by an industrial standard for automotive control software [1], most software components in the basic software layer correspond to the electronic architecture of the control system, such as microprocessors, LSIs, and network protocols. The variability of basic software is easily visible and the relationship between the variability and variation point is simple in this example. On the other hand, a feature of the application layer calculates engine phenomena, i.e. intake airflow, fuel combustion, and exhaust gas. These phenomena are correlated to many kinds of physical components such as engine size, number of cylinders, valves, fuel injectors, and fuel type. Moreover, there are some nonfunctional requirements that cause variability such as exhaust gas regulation by different countries and drivability. Therefore, the features in the application layer have much invisible variability, and the correlations between the feature and the variability are complex. Thus, we apply FAVE to extract the variability of a feature in the application layer and the mappings to software components of the feature.

An overview of the case study is shown in Table 1. The repository has data about sixteen products that were released for different vehicles. We extracted one control subsystem that consists of forty-nine software components. Each software component is responsible for a fine-grained feature of the engine or the control system itself. To compute the factors, we applied R [5] and configured the parameter, as shown in Table 2.

5.2 Analyzing Variability

We analyzed the variability by following the process in Fig. 3.

5.2.1 Converting to Change Vector

First, we converted the release histories to the change vectors. We obtained fifteen change vectors from sixteen releases of existing products from the software repository. The release of software corresponds to different vehicles in the market.

Due to a company confidentiality issue, we are unable to disclose the organization of the software repository and change vectors.

Table 3. Examination result for # of factors

# of factors	1	2	3	4
p-value	1.43E-24	5.74E-16	5.39E-8	0.0105
X squared	301.1	220.2	141.7	77.15
Cumulating	0.229	0.359	0.501	0.594

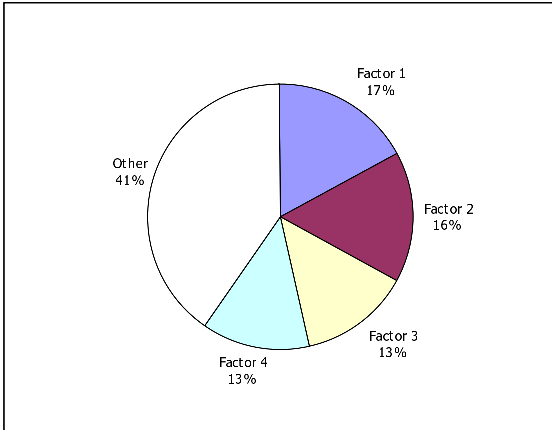


Figure 7. Factor proportion

5.2.2 Factor Analysis

- Number of factors

  Next, we examined the number of significant factors of the product line. We applied the chi-squared test and checked the cumulative variable of the factors. The result of the examination is shown in Table 3. When the number of factors is four, p-values of the chi-squared test are larger than 0.01 and the cumulating value is 0.594. Therefore, we defined the numbers as ”4”.

  The percentages of factors are shown in Fig. 7. The other than the factors indicates specific changes for each change history. From the viewpoint of the SPL, this portion corresponds to product-specific modifications.

- Factor 1

  Next, we define the variabilities for each factor based on the factor loadings and scores. Because of the page limit, we explain about factor 1 and 2 case only.

  The loading of factor 1 is shown in Figure 8. Change vectors F, H, and J correlate with factor 1. The experimental result suggests that variability relating to factor 1 has occurred in change vectors F, H, and J.

  The score of factor 1 is shown in Figure 9. Some software components have a factor score significantly higher than 1.4, and the other components have a low factor score. The experimental result suggests that

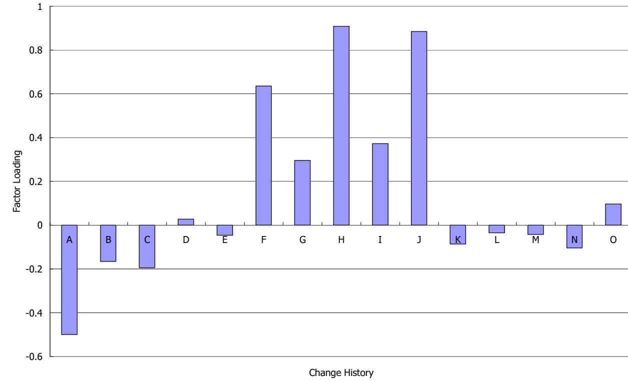


Figure 8. Factor loading (factor 1)

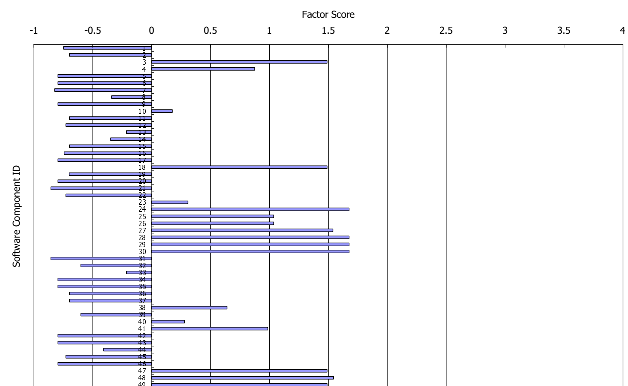


Figure 9. Factor score (factor 1)

software components No. 3, 18, 24, 27, 28, 29, 30, 47, 48, and 49 are correlated with each other. This means that there is a set of software components that have been changed together in the change vectors and FAVE extracted the set automatically.

Finally, we define the variability of factor 1. We select software components No. 3, 18, 24, 27, 28, 29, 30, 47, 48, and 49 as the variation point of the variability based on the factor score. Then, we check the features of the components and suppose what kind of variability is related to an abstracted feature. From the control specification point of view, software components No. 24, 27, 28, 29, and 30 are strongly related to the variable valve timing control feature and the others are related to the phenomenon of intake airflow. Therefore, we checked how the products are modified in the change histories F, H, and J, and then found that the valve system was modified from the fixed type to the variable type and from the variable type to the fixed type. Therefore, we defined factor 1 as the variability of ”Variable Valve Timing Control” and the selected software components as variation points.

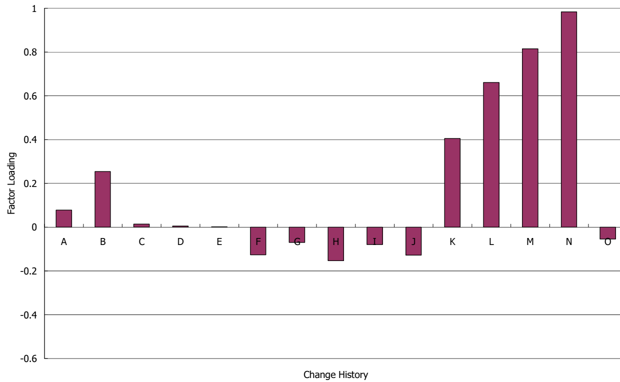


Figure 10. Factor loading (factor 2)

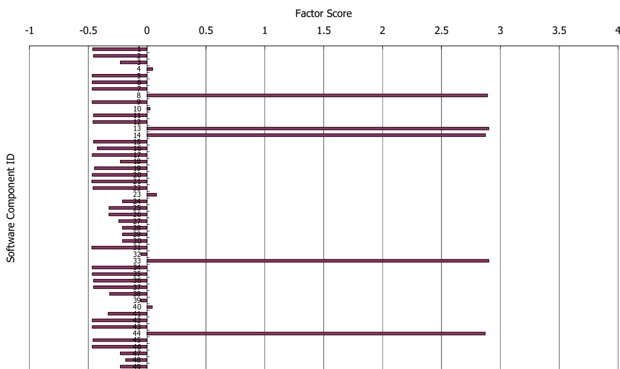


Figure 11. Factor score (factor 2)

- Factor 2

  The loading of factor 2 is shown in Figure 10. Change vectors L, M, and N correlate with factor 2. The experimental result suggests that variability relating to factor 1 has occurred in the change vectors L, M, and N.

  The score of factor 2 is shown in Figure 11. Software components No. 8, 13, 14, 33, and 44 have a factor score significantly higher than 2.8, and the other components have a low factor score. The experimental result suggests that software components No. 8, 13, 14, 33, and 44 are correlated with each other. Again, this means that there is a set of software components that has been changed together in the change vectors and FAVE extracted it correctly.

  We select software components No. 8, 13, 14, 33, and 44 as the variation point. All of them are strongly related to the ignition timing control. Therefore, we checked how the products are modified in the change histories L, M, and N, and then found that the basic functionality of the ignition timing control was updated in the histories. Therefore, we defined factor 2 as the variability of "Ignition Timing" and the selected software components as variation points.

5.2.3 Result

The detected variability in the case study is shown in Table 4.

5.3 Evaluation

We confirmed the result of the analysis with senior engineers in the business division to evaluate the practicality of FAVE.

The detected variability is the actual variability of the product line that has occurred in the product release history that we analyzed. FAVE detects the variability that causes modifications for the software components. This means that we can focus on only the detected variability and omit the remaining possible variability at least for the existing products. Moreover, FAVE detects variability that the engineers have not expected. For example, they thought that modifications related to the fuel type (factor 3) had a very minor effect on the software. As FAVE detected, the modification caused a significant effect on some of the software components and its proportion ratio was 13%. Note that the sample data is very important and the census survey is desirable.

The detected variation points that relate to software components are also relevant. Even for the expert, defining an exact set of software components that are modified by a variability is very difficult for an engineer because of the complexity and non-linearity of the physical phenomenon of the engine system. FAVE detects software components that match the experience of the expert successfully. This means that FAVE has potential to extract tacit knowledge implemented in the existing products, and we could use that as explicit knowledge. Moreover, FAVE detects some software components that we did not expect as variation points. Some software components suggested by the experimental result looked irrelevant to the variability. However, we understood that the component should be included to the variation point, after we analyzed the specification of software components. This means that FAVE may extract variation points that we have not noticed yet.

Variability 3 (Fuel type) and 4 (Intake valve) contain software component No. 4 as one of their variation points. This means that the variability in the legacy software was not orthogonal to each other. To improve modularity of the variability, software component No. 4 should be refactored. For example, we can divide the software component into two components in which one is related to Fuel type and the other is related to the Intake valve. FAVE detects the variability itself and overlap of the variability that should be refactored in the future.

In the experimental application, people performed only defining the number of significant factors and defining the

Table 4. Variability of experimental application

Factor ID	1	2	3	4
Variability	Variable valve timing	Ignition control	Fuel type	Intake valve
Variation points (SWC ID)	3, 18, 24, 27, 28, 29, 30, 47, 48, 49	8, 13, 14, 33, 44	4, 10, 23, 32, 39, 40	4, 25, 26, 41
Proportion	0.17	0.16	0.13	0.13

variability. FAVE will reduce the overhead for introducing SPL approach.

6 Conclusion

In this paper, we proposed an approach to suggest variability candidates across existing software products by analyzing the change history. We applied a factor analysis technique to analyze variability of the existing products and call our approach "FAVE: Factor Analysis based Variability Extraction." We apply the factor analysis to changes between existing products and detect the co-change patterns that may indicate the variability of the product line.

In the case study of the automotive engine-control system, we detected four variabilities and their variation points from the change history. The detected variability corresponds to the variability from the requirement viewpoint.

Clearly, this work is in its initial stage. We are currently exploring several extensions to it.

We used a part of a software product as an example for the experimental application. We hope that FAVE can be used for a whole software product, and there are many technical issues. For example, we will need to visualize the results to be understandable for domain experts. In the future, we will analyze a larger data set and study appropriate solutions.

Variability analysis is an important step, but not the goal. We need to refactor the analyzed software components so that the components can be reused as core assets of the product line. The refactoring of the detected variability is the key challenge for migrating the software product line.

FAVE only detects the existing variability. To introduce variability from a roadmap of future products, we need to integrate our approach and requirement-oriented variability analysis.

The products release history contains information about variability that occurred in the past. The history may help us to understand its variability.

References

[1] AUTOSAR. AUTomotive Open System ARchitecture. *http://www.autosar.org/*. visited on Jan. 18, 2008.

[2] J. Bayer and et al. PuLSE: A methodology to develop software product line. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, pages 122–131, May 1999.

[3] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[4] M. Fischer and et al. Mining evolution data of a product family. In *Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05)*, pages 1–5, 2005.

[5] R. Foundation. The R project for statistical computing. *http://www.r-project.org/*. visited on Jan. 18, 2008.

[6] I. John. *Software Product Lines*, chapter Capturing Product Line Information from Legacy User Documentation, pages 127–160. Springer, 2006.

[7] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets - a case study. In *Proceedings of Software Product Lines: 9th International Conference (SPLC 2005)*, pages 45–56, 2005.

[8] C. Lai and D. Weiss. *Software Product Line Engineering*. Addison Wesley, 1999.

[9] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *Proceedings of Software Product Line Conference 2007 (SPLC2007)*, pages 151–160, 2007.

[10] K. Pohl, G. Bockle, and F. V. D. Linden. *Software Product Line Engineering: Foundations, Principles And Techniques*. Springer-Verlag New York Inc, 2005.

[11] M. Steger and et al. Introducing pla at bosch gasoline systems : Experiences and practices. In *Proceedings of Software Product Lines: International Conference (SPLC 2004)*, pages 34–50, 2004.

[12] C. Tischer, A. Mueller, M. Letterer, and L. Geyer. Why does it take that long? Establishing product lines in the automotive domain. In *Proceedings of Software Product Line Conference 2007 (SPLC2007)*, pages 269–274, 2007.

[13] K. Yoshimura, D. Ganesan, and D. Muthig. Defining a strategy to introduce software product line using the existing embedded systems. In *Proceedings of 6th ACM & IEEE Conference on Embedded Software (EMSOFT06)*, 2006.

[14] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE - Factor analysis based approach for detecting product line variability from change history. In *Proc. of 5th Workshop on Mining Software Repository (MSR2008)*, 2008. (to appear).

[15] T. Zimmermann and et al. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 429–445, 2004.

The Linux Kernel Configurator as a Feature Modeling Tool

Julio Sincero and Wolfgang Schröder-Preikschat
Department of Computer Science 4
Friedrich-Alexander University Erlangen-Nuremberg
{sincero,wosch}@cs.fau.de

Abstract

In order to contribute to the understanding of how the SPL community and the open source community can benefit from each other, we present the Linux Kernel Configurator (LKC). We describe its capabilities and explain how it can be used for the design of feature models.

1. Introduction

A software product line (SPL) is a set of software components that can be composed in order to deliver a specific product. The scientific community has proposed a variety of techniques to support the development of SPLs, *feature modeling*, *product line scoping*, *feature implementation techniques*, *variability management*, among others. These approaches aim at producing flexible software architectures, reducing time-to-market, enabling substantial code reuse, and, consequently, providing high-quality software products. The literature has plenty of study cases showing the benefits of the adoption of such techniques.

Different sectors of the software industry have been adopting the SPL approach in their development process. However, another increasingly interest is the use of *open source* software. This can be motivated by many reasons, but mainly, due to cost factors and the attested quality of many open source projects. In addition, it has been shown that some open source projects, due to its technical implementation, can be considered to be SPLs [8], even though during its development process no methods proposed by the SPL community were used.

Therefore, it is clear that both communities share interests and goals. We believe that for a better understanding of how these communities could benefit from each other, technical issues should be discussed. This paper presents the *variability management* employed by the Linux Kernel. We present the open source tool that is used for this task, and also, how it can be adapted to be used by the SPL community as a *feature modeling* tool.

Motivation

We are specially interested in addressing the configuration of non-functional properties (NFPs) in SPLs[9, 5]. We believe that information about NFPs should be provided during feature selection so that the application engineer can be aware of the impact of a determined feature on the final product. Therefore, our idea was to extend a feature modeling tool in order to appropriately present this information that cannot be accommodated in the *feature diagram*, as it can be in the form of graphs or charts.

Nevertheless, we were not able to find any open source feature modeling tool to bring our extensions, and also, our goal was not to develop one from scratch. As the Linux Kernel configurator is open source and very flexible, we decided to test if it could meet our needs. It turned out that it could be easily used as a feature modeling tool. In this paper we demonstrate how to design features models with it. [1]

2. The Linux Kernel Configurator

The Linux Kernel Configurator (LKC) is a tool that is delivered within the Linux Kernel in order to enable its configuration (feature selection) . Its first prototype was proposed in 2002, the current version is 1.3, and as the Linux Kernel, it is released under the GNU General Public License.

2.1 A Little bit of History

In 2001 the community around the Linux Kernel started to show dissatisfaction with the kernel configuration tool, back then known as *configuration menu language* (CML1). With the growth of the kernel, the configuration process was getting very complicated. The tool was responsible for selecting the capabilities to be built into the kernel, handling dependencies and providing the user interface for feature selection. Moreover, it was comprised of a mixture of code

[1] The aforementioned extensions regarding NFPs will be subject of another publication.

written in Tcl/Tk scripts, awk scripts, pearl and C, which made it hard to understand and to maintain[1].

In order to solve this problem, a configuration menu language 2 (CML2) was proposed. It was a *mini-language* designed specifically for configuring kernels. A *ruleset* describing all the available options and their dependencies can be translated into a *rulebase* that is read by the front-end in order to configure the kernel[7]. After more then two years of development, several *flame wars* on the mailing list, and many improvements over the previous system, the project was dropped and not accepted in the official kernel tree (this fact shows how restrictive and demanding is the community regarding new code being merged in the official tree). Nevertheless, the source code is still available and it is used as the configuration tool of other projects.

A couple of months after the discussions about the CML2 had finished, the LKC was proposed aiming at addressing the shortcomings of both CLM1 and CML2. According to the author, the major advantages over CML2 are: (a) it is written in C code (CML2 is written in Python which makes a Python interpreter to be delivered with the kernel) (b) a tool for the automatic converting of the CML1 configuration into the new one is included, (c) it is less complex then CML2, it does not try to address problems like facilitating the kernel configuration for non-experts as the CML2 does.

As these three points were of great importance for the linux developers, after around one year of testing and improvements, the LKC was accepted and merged in the kernel `2.5.45`.

2.2 The Linux Kernel Configurator

The LKC is basically comprised of a *parser* and a *dependency checker* that are used as the back-end. To enable the selection of *configuration options* (as defined in a configuration database), different front-ends (graphical, text-mode, command-line interactive, etc.) are provided.

A configuration database is the collection of *configuration options* organized in a tree structure. Every entry has its own dependencies that are used to determine its visibility, any child entry is visible only if its parent entry is also visible. An entry either defines a *configuration option* or is used to organize them[10].

The configuration file is a text file containing the entries which must follow a strict syntax. The configuration database is built as a set of *entries* which define *configuration options*. Line 1 of Listing 1 shows[2] an entry definition, it starts with the keyword `config` and is followed by its name. The next lines of an entry are used to define its attributes, which can be the following:

[2]this listing is an excerpt of the GPL [6] feature model designed with the LKC language

type define the type of an entry, they can be `boolean`, `tristate`[3], `string`, `hex` and `integer`. An example is shown on line 2 of Listing 1

input prompt is the visual name of the configuration option that is displayed to the user during configuration. On line 1 the actual configuration name is defined as (`GPL`) which will be used in the generated configuration file, however, the user will see during configuration the name `ROOT` as shown on line 2.

default value is assigned to the configuration symbol if no value was set by the user. An example is given on line 17.

dependencies define the requirements of the menu entry. They can simply define the entry depending on a single configuration option, as shown on line 6, or can be in the form of logical expression using primitives like && (logical and), || (logical or), as shown on line 18.

reverse dependencies are used to force the lower limit of the value of another symbol. As shown on line 3, if the symbol `GPL` is selected, the symbol `M1` will automatically be selected as well.

numerical ranges limit the range of possible input values for `integer` and `hex` symbols.

help text defines the *configuration option* help text to be shown during configuration. Examples are shown on line 21 and 31.

Listing 1. LKC language

```
1  config GPL
2    boolean "ROOT"
3    select M1
4
5  choice
6    depends on GPL
7    prompt "Graph Type"
8
9    config DIRECTED
10     boolean "Directed"
11
12   config UNDIRECTED
13     boolean "Undirected"
14 endchoice
15
16 config NUMBER
17   default y if GPL
18   requires (BFS || DFS)
19   boolean "Number"
20   ---help---
21   Assigns a unique number to each
22   vertex as a result of a graph
```

[3]the boolean type can be assigned to `yes` or `no`, the tristate type allows an extra value (`m`) which means that the configuration option should be included, however, as a separate module.

```
23     traversal.
24
25   config CC
26     depends on GPL
27     requires (BFS || DFS)
28     requires UNDIRECTED
29     boolean "Connected Comp."
30     ---help---
31     Computes the connected components
32      of an undirected graph, which are...
```

A configuration database, which defines the valid configurations that can be derived with the front-ends, is created using the *entries* and the *attributes* as described above.

Moreover, in order to provide a better organization of the entries in the configuration tree that is displayed to the user, the following constructs are allowed:

menu Entries defined between the keywords `menu` and `endmenu` are grouped together and displayed in a separate window. It may also have an attribute `prompt` to name the groups of entries. An example is given between lines 5 and 14.

choice Only one entry of those defined between the keywords `choice` and `endchoice` can be selected if its parent entry is also selected.

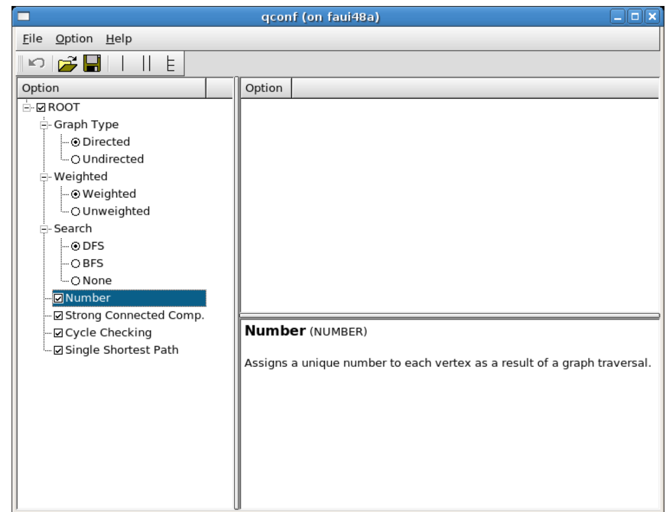
3. Feature Modeling with the LKC

The main contribution of this paper is to show how to design feature models using the LKC *configuration language* described in the previous section. After the introduction of feature models by Kang et. al.[3] many extensions were proposed. In this work we will concentrate on the basic syntax allowing *mandatory features*, *optional features or groups* and *alternative groups*. Regarding extra feature constrains, we allow *implies* and *excludes*. This decision was inspired by the work of Benavides et. al. [2], which describes the mapping from these feature model relations to representations in the form of *constraint satisfaction problem*, *boolean satisfiability problem* and *binary decision diagrams*.

Table 3 summarizes the mappings from the LKC language to feature model relations.

Most of the mappings were relatively easy to perform. For the *mandatory* relation, the parent feature forces the selection of the child by the use of a reverse dependency (`select`). The *optional* relation is described by using a dependency between the child and the parent feature (`depends on`). The *or group* is designed by creating reverse dependencies between the children and the parent, this was done inside a menu definition in order to group the children together. The *alternative group* can be described by including configuration options (the children) inside a

Figure 1. The LKC graphical interface



`choice` definition, which has the same semantic as of *alternative group* in feature models.

Using this mapping we were able to design several feature models. So far we did not find any feature model construction that could no be modeled with the LKC. Figure 3 depicts the screenshot of the LKC graphical front-end displaying the feature model of the Graph Product Line (GPL) [6] which was proposed as a standard problem for evaluating product lines. As we have[4] an implementation of this product line where the features are implemented by means of conditional compilation, the output of the configurator could be used (it is a set of pre-processor `defines`) in the compilation process of the GPL product line.

4. Future Work

As described previously we aim at extending the LKC front-end to present information about non-functional properties and use it as our feature modeling tool, these extensions are currently being implemented. Moreover, the design of a very simple textual feature modeling language and a tool to transform it in the LKC language format is currently under development.

5. Conclusion

In order to contribute to the understanding of how the SPL community and the open source community can benefit from each other, we presented the configuration tool of

[4]generated using the Colored Integrated Development Environment (CIDE)[4]

Relation	Feature diagram	LKC
MANDATORY	P, C	**config** P **boolean** "P" **select** C **config** C **boolean** "C"
OPTIONAL	P, C	**config** P **boolean** "P" **config** C **depends** on "P" **boolean** "C"
OR	P, C_1, C_2, C_3	**menu** "P" **config** P **boolean** **config** C1 **boolean** "C1" **select** P **config** C2 **boolean** "C2" **select** P **config** C3 **boolean** "C3" **select** P **endmenu**
ALTERNATIVE	P, C_1, C_2, C_3	**choice** **prompt** "P" **config** C1 **boolean** "C1" **config** C2 **boolean** "C2" **config** C3 **boolean** "C3" **endchoice**
IMPLIES	A, B	**config** A **boolean** "A" **requires** B **config** B **boolean** "B"
EXCLUDES	A, B	**config** A **boolean** "A" **requires** !B **config** B **boolean** "B" **requires** !A

Table 1. Mapping: Feature relations to LKC language

one of the most popular open source projects. We showed how it is used to describe configuration databases, and we have also explained how to describe the semantics of feature models using its language. The feasibility of the approach presented in this work, corroborates with the assumption of similarities between the technical goals that these two communities pursue.

References

[1] The linux 2.5 kernel summit. http://lwn.net/2001/features/KernelSummit/, 2005.

[2] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. 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.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[4] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.

[5] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[6] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. *Lecture Notes in Computer Science*, 2186:10–??, 2001.

[7] E. S. Raymond. The cml2 resources page. http://www.catb.org/ esr/cml2/.

[8] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In F. van der Linden and B. Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[9] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the Configuration of Non-Functional Properties in Software Product Lines. In *Proceedings of the 11th Software Product Line Conference, Doctoral Symposium (SPLC '07)*, 2007.

[10] R. Zippel. The linux kernel configurator. http://www.xs4all.nl/ zippel/lc/.