Efficient Compilation Techniques for Large Scale Feature Models

Marcilio Mendonca[1], Andrzej Wasowski[2], Krzysztof Czarnecki[1] and Donald Cowan[1]
University of Waterloo[1], IT University of Copenhagen[2]
{marcilio,dcowan}@csg.uwaterloo.ca, wasowski@itu.dk, and kczarnec@swen.uwaterloo.ca

Abstract

Feature modeling is used in generative programming and software product-line engineering to capture the common and variable properties of programs within an application domain. The translation of feature models to propositional logics enabled the use of reasoning systems, such as BDD engines, for the analysis and transformation of such models and interactive configurations. Unfortunately, the size of a BDD structure is highly sensitive to the variable ordering used in its construction and an inappropriately chosen ordering may prevent the translation of a feature model into a BDD representation of a tractable size. Finding an optimal order is NP-hard and has for long been addressed by using heuristics.

We review existing general heuristics and heuristics from the hardware circuits domain and experimentally show that they are not effective in reducing the size of BDDs produced from feature models. Based on that analysis we introduce two new heuristics for compiling feature models to BDDs. We demonstrate the effectiveness of these heuristics using publicly available and automatically generated models. Our results are directly applicable in construction of feature modeling tools.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Computer-aided software engineering (CASE)

General Terms Design

Keywords Model-driven development, software-product lines, formal verification, Configuration, feature modeling

1. Introduction

Generators and components support the creation of systems within system families. A system family is a set of systems sharing enough common properties to warrant basing their development on a common set of reusable assets, such as frameworks, components, and generators. Building such assets requires understanding both the common features and the varying features of systems within a family. For example, all e-commerce systems are likely to provide common features such as catalog browsing and product checkout. However, the systems may differ in several respects, e.g., whether they support selling physical products or electronic products or both and whether they allow guest or registered checkout or both.

Feature modeling is a technique for representing the common and variable features of systems in a system family. A *feature model* is a hierarchy of mandatory, optional, and alternative features with possibly additional constraints, such as implications between pairs of features [19]. Feature modeling is used in system family scoping, i.e., deciding which features should be supported by the common assets and which not, identifying architectural variation points, and in system configuration [10]. Feature models can directly represent a class of domain-specific languages that are focused on configuration. Systems can be specified as configurations of features and such specifications can be used as input to code generators, e.g., [14], or to configure requirements and design models [12] or components [6].

Feature models have been semantically related to propositional logic [5, 13]. The translation of feature models into logic representations has allowed the use of reasoning tools for automated feature model analyses [4], such as consistency checks and finding dead features, refactoring [2], reverse engineering [13], and interactive configuration [25]. All of these applications require an efficient representation of the configuration space of the features. Binary Decision Diagrams (BDDs) [9, 22] are one such representation, which supports efficient logical tests and interactive guidance algorithms [18]. Interactive configuration is a process of selecting a particular variant out of those represented by a model. The process is interactive since it includes user steps, such as selecting and eliminating features, and the machine responses, such as selecting implied features and excluding incompatible features. The interactive guidance in this context is provided by calculating so-called *valid domains*, i.e., possible assignments of features given the current state of the

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

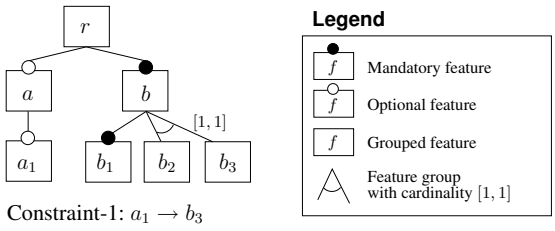


Figure 1: A feature model

system, and propagating information whenever new choices are made.

Efficiency is extremely important for some of the above applications, in particular for detecting refactoring opportunities and for interactive configuration. These two are normally performed as a part of each interaction and thus should guarantee response times within milliseconds. Since the response time of most of standard algorithms on BDDs requires time proportional to the size of the BDD [3], it is desirable to devise technologies that can decrease this size as much as possible. One such approach is a topic of our present paper.

Each BDD has a fixed variable ordering associated. This ordering has a dominant influence on its size. A bad order can be seriously detrimental leading to excessive memory use, often beyond capabilities of typical workstations. On the other hand a very good order can dramatically compress the BDDs (down to kilobytes!), enabling extremely fast processing with interactive algorithms. It is thus natural that quite a few researchers in various domains have investigated the ordering minimization problem. In this paper, we approach this problem by proposing efficient ordering heuristics for the feature modeling domain. Experimental results show that the proposed heuristics allow for efficient compilations of feature models with up to 2,000 features.

We proceed with Section 2 providing a short background on feature models, and Section 3 introducing the variable ordering problem. State-of-the-art solutions are reported in Section 4. Section 5 documents design decisions in choosing a new reordering heuristics for feature models, followed by Section 6 presenting the heuristics, experimental analysis (Section 7) and conclusion (Section 8).

2. Feature Models

A *feature model* consists of (i) a *feature tree* and (ii) possibly one or more *extra constraints*, which are propositional formulas over features. Fig. 1 depicts a sample feature model. Its feature tree (top left) has a root feature r, mandatory features b and b_1, optional features a, and a_1, and an exclusive-OR group containing grouped features b_2 and b_3. The implication $a_1 \rightarrow b_3$ labeled *Constraint-1* is the extra constraint.

A feature model denotes a set of *legal configurations*. A *legal configuration* is a set of features selected from the feature model according to its semantics. The set of legal configurations is given by a conjunction of the extra constraints with a propositional formula that is systematically constructed from the feature tree [5, 13]. The formula is a conjunction of (i) the root feature, (ii) an implication from each child feature to its parent, (iii) an implication from each feature with a mandatory child to that child, (iv) an implication from a parent with an inclusive-OR (exclusive-OR) group to a disjunction (pairwise mutual exclusion) of the group members. Applying this derivation to the sample feature tree in Fig. 1 yields, after some simplifications, the formula $r \wedge b \wedge b_1 \wedge (a_1 \rightarrow a) \wedge (b \rightarrow b_2 \,\mathsf{xor}\, b_3)$.

Next, we provide definitions that are used throughout the paper:

DEFINITION 1. *The extra constraint representativeness (ECR) is the ratio of the number of variables in the extra constraints (repeated variables counted once) to the number of variables in the feature tree.*

ECR for the feature model in Fig. 1 equals $\frac{2}{7} \simeq 0.28$.

DEFINITION 2. *For features f_1, ..., f_n their* lowest common ancestor*, written LCA$(f_1, \ldots, f_n)$, is their shared ancestor that is located farthest from the root (where a feature is an anscestor of itself).*

For features of *Constraint-1* we have $\mathrm{LCA}(a_1, b_3) = r$.

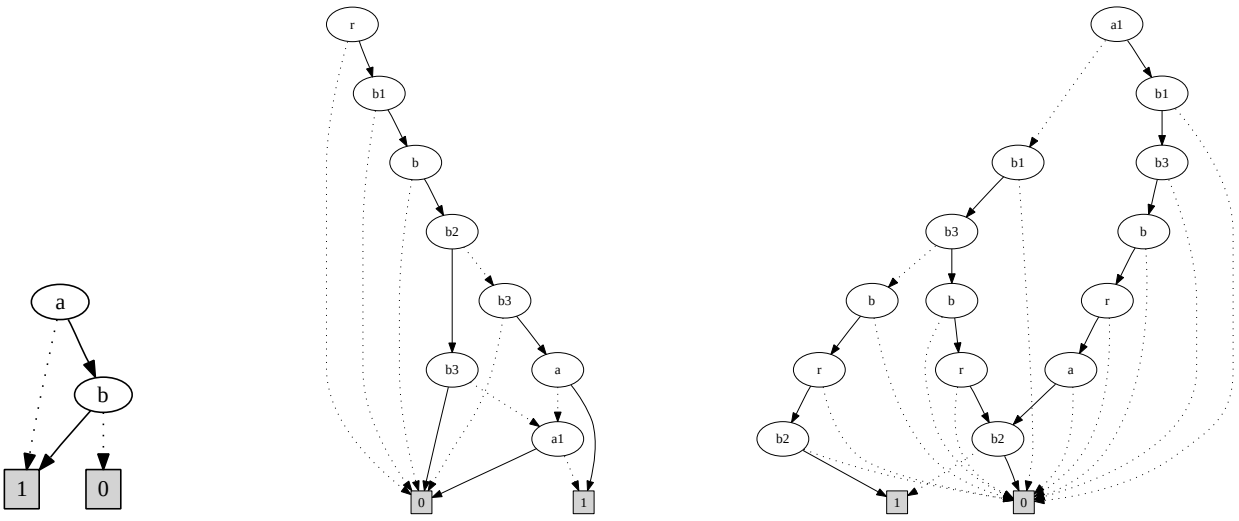
DEFINITION 3. *Given f = LCA$(f_1, \ldots, f_n)$, the* roots *of features f_1, ..., f_n, written Roots$(f_1, \ldots, f_n)$, is either the set {f}, if f has no children, or the subset of f's children that are ancestors of f_1, ..., f_n.*

In our example $\mathrm{Roots}(a_1, b_3) = \{a, b\}$, since features a and b root the subtrees containing a_1 and b_3 respectively.

3. BDDs and The Variable Ordering Problem

decision diagram (BDD) [9, 3] is a concise representation of a Boolean function. BDDs are directed acyclic graphs (DAGs) having exactly two *external nodes* representing constant functions 0 and 1, and multiple *internal nodes* labeled by variables. For instance, Fig. 2a depicts a BDD for formula $(a \rightarrow b)$. Each internal node has exactly two outgoing edges representing a decision based on an assignment to the node variable: the *low-edge* (a dotted line in the figures) represents the choice of false, while the *high-edge* (solid) represents the choice of true. A path from the root to an external node represents an assignment of values to variables. For example the rightmost path in Fig. 2a represents a (non-satisfying) assignment $[\, a \mapsto 1, b \mapsto 0 \,]$. The paths terminating in the external node 1 (respectively 0) represent satisfying (respectively unsatisfying) assignments.

A BDD is *ordered* if every top-down path in the DAG visits the variables in the same order. In a *reduced* BDD any two nodes differ either by labels or at least by one of their children (*uniqueness*), and no node has both edges pointing to the same child (*non-redundancy*). Notice that the three



Variable order: (a) $a < b$ (b) $r < b1 < b < b2 < b3 < a < a1$ (c) $a_1 < b_1 < b_3 < b < r < a < b_2$

Figure 2: (a) A simple BDD; (b-c) BDDs for the model in Fig. 1 with two different variable orders

BDDs in Fig. 2 are both reduced and ordered. We shall use the term BDD as a synonym for ROBDDs from now on.

During the last two decades BDDs have been widely applied to address large scale combinatorial problems in logic synthesis, verification, configuration, constraint satisfaction and optimization. Off-the-shelf BDD libraries are freely available (e.g. JavaBDD, Buddy, CUDD). What makes BDDs so appealing is polynomial time algorithms for applying Boolean connectives and constant time satisfiability and equivalence checks (recall that these are generally NP-hard). Crucially for configuration, polynomial algorithms for computing valid domains are known [16, 17].

Fig. 2b presents a BDD for the formula $r \wedge b \wedge b_1 \wedge (a_1 \rightarrow a) \wedge (b \rightarrow b_2 \,\mathsf{xor}\, b_3) \wedge (a_1 \rightarrow b_3)$, which corresponds to the model of Fig. 1. The BDD contains 8 internal nodes, 2 external nodes, and 3 satisfying paths, each representing one or more solutions.

A major drawback of BDDs is their high sensitivity to variable ordering. For an illustration of the problem, consider the BDD in Fig. 2c representing *the same formula* as Fig. 2b, but with another order. While the original BDD (Fig. 2b) had only 10 nodes, the new one contains as much as 16 nodes—60% more! In the worst case this difference is exponential, which can translate to millions of nodes in practical applications. Unfortunately, finding an optimal variable order, which minimizes the size of a BDD, is an NP-hard problem [7, 22]. For this reason it is typically approached by heuristic algorithms. Heuristics exploit specifics of the problem domain in order to compute good orders efficiently. Typically research communities applying BDDs develop such heuristics for their domain. In this paper we investigate the problem for the feature modeling domain, with the goal of enabling BDD-based feature modeling tools to handle very large models.

4. A Survey of Variable Ordering Heuristics

A pervasive goal of all the ordering heuristics is placing variables that are combinatorically related close to each other in the ordering. This task is nontrivial. Dependencies between variables often interfere: optimizing the placement of a variable with respect to one dependency often decreases the quality of the ordering with respect to the others.

Variable ordering heuristics can be categorized into *dynamic* and *static*. Dynamic heuristics reorder the variables on-the-fly during construction and manipulation of a BDD, usually exploiting library's garbage collection cycles. Static heuristics compute a variable order off-line, which is then applied once to construct and analyze the BDD.

4.1 Static Heuristics

BDDs have been very successful in synthesis and analysis of digital circuits. Similarly to a BDD, a circuit represents a Boolean function, and there exist direct translations between circuits and BDDs in either direction. Since the efficiency of verification strongly depends on the size of the BDD used, it is not surprising that the variable ordering problem has been deeply studied for the circuit domain.

Feature models can be easily translated to Boolean circuits, which enables the use of existing ordering heuristics from that domain. Since Boolean connectives directly correspond to gates of the circuit, one can translate feature models in a syntax directed way. We have implemented this translation to basic circuits with AND, OR, and NOT gates and evaluated the usefulness of circuit heuristics described below for the compilation of feature diagrams. The translation

is linear for all the feature model elements except for xor-groups, for which is it is quadratic in the size of the group. In our evaluations, the translation produced circuits 3 to 10 times larger than the corresponding feature model.

Fujita's Heuristic. Fujita-DFS [15] is a heuristic that traverses the circuit from the output to the inputs (which correspond to variables) in a depth-first search (DFS) order. During the traversal, inputs connected to two or more gates are placed first in the generated variable ordering in the hope that the remaining nodes in the circuit will form a tree-like structure for which a standard DFS produces good variable orderings. Since a circuit is a directed-acyclic graph (DAG) with a single output, if nodes connected to many other nodes are removed from such a rooted DAG, the remaining structure approximates a tree. Fujita-DFS proved to generate good orderings for some circuit benchmarks, e.g. ICAS-85 [8].

Level Heuristic. The *level* heuristic [21] assigns the *depth level* to each circuit node, which is the length of the longest path from that node to the output. Subsequently, the inputs are sorted in decreasing order of levels to produce the final order. The level heuristic performs particularly well for multi-level circuits in which the outputs of a sub-network serve as inputs to the next subnetwork in the chain.

FORCE Heuristic. FORCE [1] is a domain-independent static heuristic for variable ordering. The heuristic is applied to a CNF formula and uses a measure called *span* to assess quality of placement for related variables. Given a pair of variables its span is defined to be their distance in a given variable ordering. The span of a clause is the maximum span of all pairs of variables occurring in the clause. Finally, the span of a CNF formula is the sum of spans of all its clauses.

FORCE begins with a random variable ordering and through successive steps attempts to minimize the formula span by moving variables near each other. At each iteration a new order is produced, which serves as input for the next iteration. It stops when the span value no longer decreases.

In order to apply FORCE, we implemented a simple CNF translation algorithm that traverses the feature tree in DFS and generates CNF clauses for each parent-child and feature group relation, as described in Section 2.

4.2 Sifting

Sifting [26, 22] is a popular domain-independent *dynamic* heuristic implemented in most BDD libraries. Unlike a static heuristic, sifting operates dynamically by trying to reduce the size of an already *existing* BDD on demand or on-the-fly; for example during garbage collection cycles. The main advantage of sifting is that it can enable the construction of BDDs that cannot be built with static heuristics.

Sifting is a local search algorithm. It swaps variables in the BDD if this leads to an improvement of the BDD size. Despite its merits, sifting has a serious drawback. The heuristic can be extremely slow in practice. In fact, we observed running times of over an hour for tasks that could be performed in a few minutes by good static heuristics. This is primarily caused by the fact that unlike FORCE a swap in a variable ordering requires a modification of the existing BDD to obey the new ordering.

5. Variable Orders & Feature Models

Many heuristics adopt the rationale of identifying and shortening the distance of dependent variables as a means to produce good variable orders. For instance, in the Level heuristic connected variables share the same level in the circuit. Fujita's heuristic uses a DFS traversal to identify connected variables in a circuit. As we mentioned before, span is the measure used by FORCE to approximate connected variables in a CNF formula. Based on this observation, we characterize the problem of ordering BDD variables in our domain as the problem of identifying related variables in feature models and producing variable orders that minimize the relative distance of such variables. What makes the problem particularly challenging is the fact that the relations in the extra constraints usually connect independent branches in the feature tree. This causes good orders for the feature tree to be extremely inefficient for the extra constraints, and vice-versa. In addition, the larger the ECR (see Definition 1 in Section 2) of a feature model the harder is to find a good order that suits both the feature tree and the extra constraints.

One way of obtaining an ordering heuristic is to compile a feature model into an intermediate representation such as a CNF formula or a circuit and use available heuristics to process the ordering. However, this approach would completely ignore the domain knowledge. For instance, the variables in the feature tree are arranged hierarchically in a tree, for which simple traversals produce good orders. At the same time, as will be seen later, such arrangements are obscured in a CNF or circuit representation, which prevents the respective heuristics from exploiting them.

Given the BDD variable ordering problem in configuration, we pose the following hypothesis: *A significant reduction in the size and construction time of BDDs representing feature models can be achieved if the structural characteristics of the models are exploited to order the BDD variables.*

In the following, we consider factors that influence development of new heuristics for variable ordering in the feature

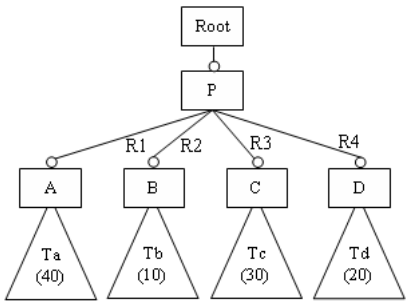
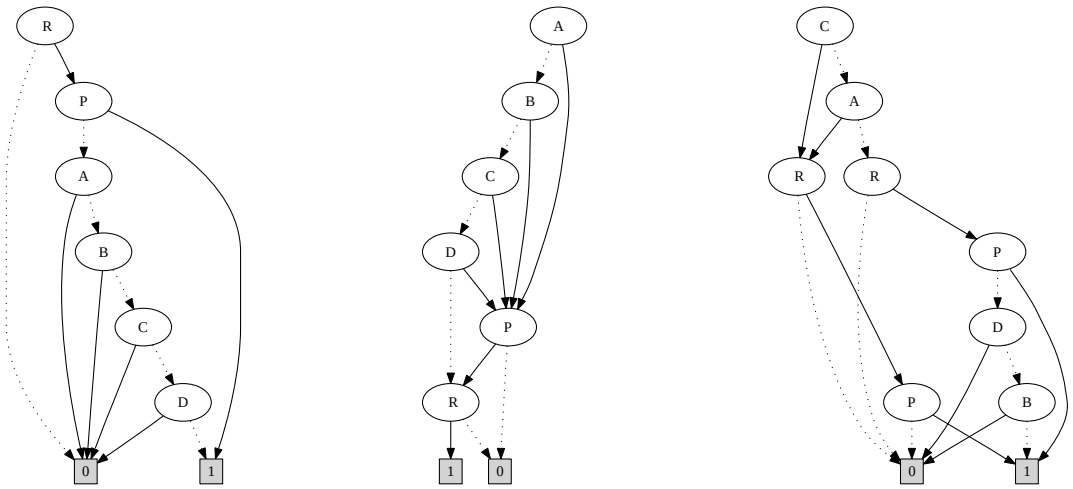


Figure 3: Feature P and children A, B, C, D



(a) Pre-order (b) Post-order (c) Average-order

Figure 4: BDDs for various traversals of the feature tree

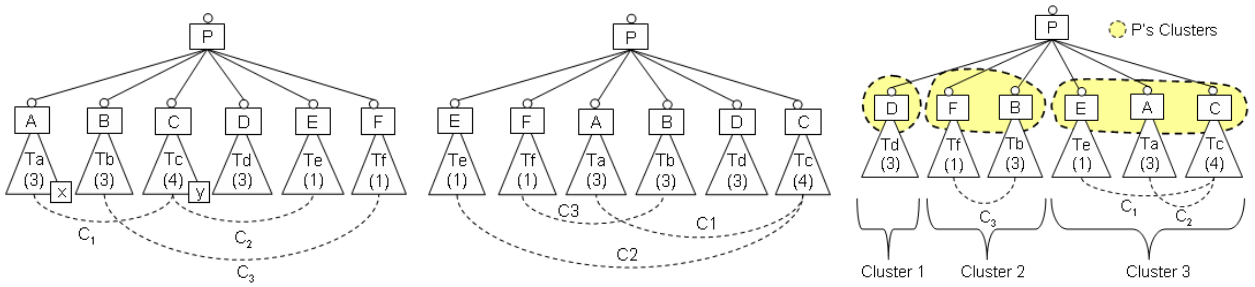
modeling domain. These considerations are then exploited in the next section when we propose such heuristics.

Structure of relations in the feature tree is explicit. The feature tree defines the variables in the feature model and specifies most of its relations. Hence, good orderings for the feature tree are generally effective for the feature model. Since relations in the feature tree are well-known and follow a hierarchical arrangement, compact structural patterns can be identified for BDDs using simple traversal algorithms.

Mandatory features disturb the analysis. Feature models allow the specification of mandatory features which might improve system family documentation but play no role in variability analysis. Mandatory features can be eliminated from the analysis as they represent binary bi-implications and hence can be automatically inferred from other features. A simplification algorithm safely removes mandatory features from the feature tree and updates all references to such features both in the feature tree and in the extra constraints, while preserving the core semantics of the model. The reduction of the number of features in a feature model can significantly reduce the size of BDDs since each feature potentially corresponds to multiple BDD nodes.

Parent-child relations define the connected variables. Feature tree constraints are expressed in terms of ancestral relations and groups. Our experiments have revealed that minimizing the distance between sibling features in groups does not improve BDD sizes. Therefore, we only consider parent-child relations to identify connected variables. Fig. 3 shows an example of four parent-child relationships involving a feature P and its children A, B, C, and D. Since all five features are optional, relations R1, R2, R3 and R4 represent binary implications (child $\rightarrow$ parent). The goal of a good heuristic for the feature tree should be to minimize the relative distance between P and each of its children in the variable order produced. Excessive minimization in one branch of the tree might cause poor minimization in others. For instance, one might decide to order variables P, A, B, C, and D in a straight sequence. However, by doing so features B, C and D are placed in between A and its children increasing their relative distance. In fact, if this strategy is applied recursively in the feature tree, BFS traversal of the feature tree is implemented, which is an extremely poor ordering.

Pre-order produces good BDD patterns. DFS traversals produce good variable orders for feature trees. However, much can be done in terms of minimizing the distance between parent and children features than pre-order. For instance, a better approach would be to place the parent node in an average distance to its children. Surprisingly, this produces BDDs with chaotic structures that in many cases are larger than one expects. We observed that the placement of parents prior (pre-order) or after (post-order) their children often produced compact BDD structures. Fig. 4 shows three BDDs for features Root, P, A, B, C and D from Fig. 3. A variable order for a pre-order traversal of the feature tree is shown in Fig. 4a (R indicates the root feature). A BDD of size 6 is shown and a very compact structure is observed for pre-order, e.g., if P is true the BDD evaluates to true no matter the values of its children. Conversely, if P is false, whenever A, B, C, or D are true, the BDD evaluates to false. Post-order also produces compact patterns (Fig. 4b); however, the BDD structure contains a much higher number of paths to the one terminal. If P is placed between its children and R is placed near P (referred to as average-order in Fig. 4c) the size of the BDD increases to 8 nodes even though the relative distance between P and its children is reduced. Thus, we adopt pre-order as the reference variable ordering imple-



(a) Natural Pre-order (b) Sorted Pre-order (c) Clustered Pre-order

Figure 5: Three different arrangements for child features A, B, C, D, E, and F

Table 1: Variable distances for pre-order-based traversals of the feature tree

Feature Tree Traversals	Variable Order	Feature Tree (FT) and Extra Constraint (EC) Variable Distances									
		FT Var. Distance	EC Shortest Distance				EC Longest Distance				EC Average Var. Distance
			C1	C2	C3	Total	C1	C2	C3	Total	
Natural Pre-Order	P<A<B<C<D<E<F	67	5	5	12	22	12	10	16	38	30
Sorted Pre-Order	P<E<F<A<B<D<C	48	9	15	5	29	16	20	9	45	37
Clustered Pre-Order	P<D<F<B<E<A<C	54	5	1	1	7	10	8	5	23	15

mentation due to its simplicity and effectiveness. We refer to this ordering as *natural pre-order*.

Sorting decreases parent-child distances. A drawback of the natural pre-order is that it relies on the natural placement of nodes in the feature tree which is not necessarily good from the point of view of variable distance minimization. Consider again the feature model in Fig. 3, showing four subtrees Ta, Tb, Tc, and Td containing 40, 10, 30, and 20 features, respectively. Natural pre-order would produce the order: P < A < [Ta] < B < [Tb] < C < [Tc] < D < [Td] where [Tn] replaces the set of features in subtree Tn. Hence, the total distance between feature P and its children is 180, i.e., 1 (A to P) + 42 (B to P) + 53 (C to P) + 84 (D to P). However, if the subtrees rooted by A, B, C and D are sorted in ascending order of their size the new order would be: P < B < [Tb] < D < [Td] < C < [Tc] < A < [Ta] and the total distance of P and its children is reduced to 110. Note that pre-order is preserved, only the relative order in which children are visited has changed. We refer to this ordering as *sorted pre-order*.

Grouping dependent subtrees minimizes variable distances in the extra constraints. So far we have focused primarily on the feature tree. However, in practice feature models can have a significant number of extra constraints considerably affecting the size of the BDD. One way to take the extra constraints into account could be to group the children of a node together based on identified dependencies among their subtrees, instead of purely sorting nodes by subtree size. Fig. 5a shows a parent feature P, its children A, B, C, D, E, and F, and subtrees Ta, Tb, Tc, Td, Te, and Tf rooted by each of P's children. Three extra binary constrains are shown: C1, C2 and C3. These constraints indicate that some of the subtrees of P's children have dependencies: Ta and Tc for C1, Tc and Te for C2, and Tb and Tf for C3. Different node arrangements are shown representing the visiting order of different pre-order-based traversals: natural pre-order (a), sorted pre-order (b), and clustered pre-order (c), where the later will be explained shortly.

Table 1 shows the variable orders and the relative variable distances for the three different traversals depicted in Fig. 5. The first row shows the distances for the natural pre-order traversal. The total distance between P and each of its children is 67 (column *FT Var. Distance*). Columns *EC Shortest Distance* and *EC Longest Distance* indicate the shortest and longest possible distances for extra constraint variables for each traversal as well as the average parent-child distance, i.e., the mean of the shortest and longest distances (column *EC Average Var. Distance*). For natural pre-order, the shortest (respectively longest) distance between variables in the constraint C1 is 5 (respectively 12). In the shortest-distance case, C1 variables correspond to features *X* (see bottom-right feature on subtree *Ta* in Fig. 5a) and *C*. In the worst case, they correspond to features *A* and *Y* (see bottom-right feature on subtree *Tc* in Fig. 5a). The average total distance of all variables occurring in the extra constraints is 30 (column *EC Average Var. Distance*).

The sorted pre-order traversal (second row in Table 1) considers sorting child nodes in ascending order of the size of their subtrees. The distance between P and its children is

reduced to 48. However, since this traversal does not take the extra constraint into account a bad average distance of 37 is observed. Fig. 4b shows the new arrangement of P's children for sorted pre-order.

The third traversal, clustered pre-order, considers using extra constraint relations to decide which nodes should be visited first. Note that in Fig. 5c nodes A, B, C, D, E and F were rearranged based on the dependencies of their subtrees. Features E, A, and C were grouped together into *clusters* since constraints C1 and C2 connect their subtrees. The same is observed for features F and B because of constraint C3. Feature D is isolated as none of its descendants is referred in the extra constraints. Three clusters are shown in Fig. 5c: Cluster 1, Cluster 2, and Cluster 3. Note that the clusters have been sorted according to their size from left to right so that larger clusters are in the rightmost positions. The size of a cluster is the total number of nodes of its contained trees. The combination of these two techniques, sorting and clustering, can considerably improve the quality of orders produced by clustered pre-order traversals. In fact, while clustering enforces distance minimization of extra constraint variables, sorting aims at parent-child distance minimization in the feature tree. A slightly higher distance for parent-child variables is observed for the clustered pre-order when compared to sorted pre-order (54 against 48, respectively), but still much better than natural pre-order (67). Yet, a significant improvement on distance minimization for extra constraint variables is achieved (15 against 37 for sorted pre-order and 30 for natural pre-order).

Algorithm 1 Clustering algorithm for the feature tree

For each feature in the feature tree, group its child nodes into clusters to indicate subtree dependency

Function **Process-FT-Clusters**()

1: $F \leftarrow$ Extra constraints in CNF
2: **for** (**each** clause C of F) **do**
3: $A \leftarrow$ *LCA*(C's Variables)
4: $H \leftarrow$ Hypergraph attached to A
5: **if** ($H = NIL$) **then**
6: Create H and attach it to A
7: Add A's child nodes as vertices in H
8: **for** (**each** of A's child node N) **do**
9: Add a hyperedge $\{N\}$ to H
10: **end for**
11: **end if**
12: $R \leftarrow$ *Roots*(C's variables)
13: Merge H's hyperedges that share elements in R
14: Associate set R to the merged hyperedge
15: **end for**

Algorithm 2 *Pre-CL* recursive algorithm

O: variable order list
N: feature being visited in the feature model
S: strategy to sort cluster's internal nodes

Function **Pre-CL-Rec**(O, N, S) : { }

1: $O \leftarrow O \cup \{N\}$
2: $H \leftarrow$ Hypergraph attached to N
3: $L \leftarrow \{\}$
4: **if** ($H = NIL$) **then**
5: $L \leftarrow N$'s children sorted in ascending order of size (the size of child c corresponds to the total number of nodes in the subtree rooted by c)
6: **else**
7: Sorts H's hyperedges in ascending order of size (the size of hyperedge E corresponds to the total number of nodes in the subtrees rooted by each element $e \in E$)
8: **if** ($S = SIZE$) **then**
9: Sorts hyperedge's elements in ascending order of size (the size of element e is the total number of nodes in the subtree rooted by e)
10: **else if** (S = *MIN_SPAN*) **then**
11: use FORCE to sort hyperedge's element
12: **end if**
13: **for** (**each** sorted hyperedge E) **do**
14: $L \leftarrow L \cup \{$nodes of FT corresponding to $E\}$
15: **end for**
16: **end if**
17: **for** (**each** node P in L) **do**
18: *Pre-CL-Rec*(O,P,S)
19: **end for**
20: Return O

6. New Heuristics for Configuration

Based on the considerations previously made we propose two novel heuristics to order BDD variables in configuration: *Pre-CL-Size* and *Pre-CL-MinSpan*. The heuristics rely on pre-order traversals of the feature tree and use sorting and clustering. Since the heuristics share many implementation aspects, a single parameterized algorithm is provided. In fact, we refer to both heuristics as part of the *Pre-CL* family of heuristics as we hope that the family will gain new members in the future.

Algorithm 1 implements the clustering process discussed earlier: function *Process-FT-Clusters*. Initially, the algorithm iterates over a set of CNF clauses that represent extra constraint relations (lines 1-2). For each clause, its variables are used to identify the LCA (Definition 2 in Section 2) node *A* in the feature tree (line 3). A hypergraph *H* is attached to *A* containing a hypernode and a hyperedge for each of *A*'s children. The hyperedges indicate that each child node is initially a single cluster (lines 5-11). Next, for each CNF clause, the *Roots* (Definition 3 in Section 2) of its variables

(a subset of A's child nodes) are grouped into a cluster to indicate node dependency (line 12). The hyperedges in H are merged so that no two distinct hyperedges share any common elements (line 13). This will make dependent nodes part of the same cluster. Finally, the dependencies identified among A's child nodes are attached to merged hyperedges.

Algorithm 2 implements a recursive pre-order traversal of the feature tree guided by the clusters processed in algorithm 1. The first call to algorithm 2 takes as input an empty list (variable order), the root of the feature tree (starting point), and the strategy to sort clusters' internal nodes (either *SIZE* or *MIN_SPAN*). Each visited node is immediately added to the variable order (line 1). For each node N, its clusters are retrieved from the attached hypergraph H (line 2). Note that only LCA nodes of extra constraint relations have clusters attached to them. List L, initially empty, stores the order in which N's children are to be traversed (line 3). If N does not have clusters associated, i.e., hypergraph H is NIL, L will store N's child nodes in ascending order of the size of their subtree (lines 4-5). If N has clusters, its clusters are initially sorted in ascending order of their size. Subsequently the internal nodes of each cluster are rearranged based on two distinct strategies: if strategy S refers to constant *SIZE* the internal nodes are to be sorted based on the size of their subtrees (just as clusters were sorted). Instead, if S is *MIN_SPAN*, internal nodes are rearranged so that their relative distance is minimized (lines 7-12). Note that we use FORCE in line 11 to sort the internal nodes of clusters. A CNF formula is encoded for each cluster as follows: the internal nodes are variables and the relations attached to H's hyperdges in line 14 of algorithm 1 are the clauses. FORCE will try to put internal nodes connected to many others in an average distance to them. In line 13, N's clusters, now sorted, are traversed and the internal nodes, also sorted, are added to list L (line 13-15). Finally, L's variables are traversed in order and a recursive call to *Pre-CL-Rec* is made to address the remaining variables (features).

7. Experiments & Analysis

We conducted several experiments to evaluate the performance of the Pre-CL heuristics against the most competitive heuristics discussed in Section 4. An AMD Turion system with a 1.6 GHz processor and 1 GB RAM and running Windows XP supported the experiments. The testing tool [23] was developed in Java using JRE 1.4.2 and ran on 650 MB of dedicated memory. The JavaBDD package [27] version 1.0b2 (JFactory instance) supported BDD manipulation.[1]

The experiments used both automatically-generated and publicly-available feature models. Five feature models previously published in the literature were considered as shown in Table 2. Generated models were grouped into collections based on the ECR and the number of features. The odds for

[1] The initial size and incremental factor for the BDD node table was set to 5 million nodes and 0.2 (20%), respectively, for all test cases.

Table 2: Feature Models from Literature

Model	Number of Features	ECR[%]
Model Transformation [11]	71	0%
Weather Station [6]	18	22%
Web Portal [24]	35	25%
e-Shop [20]	213	15%
Graph Product Line [5]	16	81%

mandatory, optional, inclusive-OR, and exclusive-OR features were 25%, 35%, 20% and 20%, respectively. Extra constraint relations were generated in 2-CNF so that each clause corresponded to a individual constraint. Clause variables were selected randomly in the feature tree. In addition, extra constraint relations were modularized at levels 0, 1, 2, and 3 (15%, 30%, 50%, and 5%, respectively) in the feature tree. We say a constraint is modularized at level n if the LCA of its variables is a node at level n in the tree. All feature models were then simplified by having their mandatory features safely removed. For further details on the experiments including tool support please refer to the project website [23].

Quality of BDD Size Reduction. Table 3 shows average space and time values for five different heuristics. We do not include the results for the *level* heuristic, as it performed extremely poorly. Fifty feature models of 500 features and 20% ECR were used in the tests. Columns *Heur. Time* and *BDD Time* indicate the percentage of the total running time for producing the variable order and building the BDD, respectively. The total time in milliseconds and the size of BDDs are shown in columns *Total Time* and *BDD Size*. The *Best Results* column indicates the number of test cases in which the heuristic had the best performance among all others. Finally, column *Failures* shows the number of test cases that resulted in overflow errors.

BDD sizes for Pre-CL-MinSpan and Pre-CL-Size were significantly smaller than for any other heuristic. For instance, average reduction rates of 95% and 61% were achieved when compared to natural pre-order that ranked third. Pre-CL-MinSpan led to smaller BDDs in 84% of the cases, while Pre-CL-Size and Fuj-DFS performed best in 12% and 4% of the cases (column *Best Results*). In terms of BDD reduction, Fuj-DFS was slightly worse but still competitive with natural Pre-order. FORCE produced poor results mainly due to its random starts. BDDs for FORCE were 76 times larger, on average, than those for Pre-CL-MinSpan.

In none of the 50 test cases BDD construction failed for any of the Pre-CL heuristics. FORCE and Fuj-DFS could not complete in 10% and 4% (column *Failures*) of the test cases due to memory overflows. Pre-order had the best heuristic running time due to its very simple algorithm that performs linearly on the size of the feature tree. However, Pre-CL heuristics were not far behind, just a few milliseconds worse

Table 3: Average running times and BDD sizes for 50 feature models with 500 features and 20% ECR

Heuristic	Heur. Time [%]	BDD Time [%]	Total Time [ms]	BDD Size	Best Results	Failures
Pre-CL-MinSpan	0.9	99.1	816	5186	42	0
Pre-CL-Size	0.6	99.4	937	43036	6	0
Pre-Order	0.1	99.9	1253	111307	0	0
FORCE	55.1	44.9	29828	394595	0	5
Fuj-DFS	0.6	99.4	1246	120608	2	2

Table 4: Scalability Measures for Pre-CL Heuristics

Heuristic	Feature Tree Size[ECR%]					
	1000 [20%]	1000 [30%]	2000 [10%]	2000 [20%]	2000 [30%]	5000[10%]
Pre-CL-MinSpan						
Successes [%]	100	98	100	46	20	0
Memory Overflows [%]	0	2	0	54	80	100
Pre-CL-Size						
Successes [%]	82	72	64	20	2	0
Memory Overflows [%]	18	28	36	80	98	100

but achieved better total running times than Pre-order. Fuj-DFS also required low running times for producing variable orders. Again, FORCE did not performed well. For all test cases, FORCE required more time to produce orders than to build the BDD (55.1% and 44.9%, respectively). In a typical run, the algorithm took 96 steps to reduce an initial span of 147,153 to a minimum span of 17,361. Each step took about 0.27 milliseconds to run which led to a total running time of 27 seconds.

In another experiment, we tried to use FORCE to improve the orders produced by Pre-CL heuristics. FORCE was given initial orders produced by Pre-CL-MinSpan and Pre-CL-Size and strived for improvements based on span minimization. Despite the lower spans obtained FORCE was unable to improve the quality of Pre-CL heuristic orders for 84% of the cases. This suggests that Pre-CL heuristics already produce high quality orders.

Real Feature Models. We applied with the heuristics in Table 3 on five real feature models previously published in the literature (Table 2). The results observed mirrored those for automatically-generated feature models (see Fig. 6). Pre-CL-MinSpan and Pre-CL-Size heuristics produced the smallest BDDs in all the cases allowing an average BDD size of 275 and 354 nodes, respectively. The space reduction was substantial even when compared to the heuristic on third place (FORCE)—around 86% less nodes.

Fujita-DFS was competitive to Pre-order in most of the cases however a poor performance for the *e-Shop* feature model ranked the heuristic in the last spot. Similarly, FORCE had performance comparable to Pre-order but its

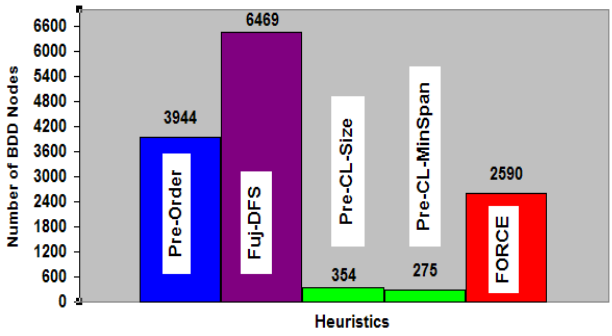


Figure 6: BDD Sizes for Real Feature Models

better order for the *e-Shop* feature model ranked the heuristic in third place.

Scalability. Table 4 shows the results of scalability tests for Pre-CL heuristics. Table columns indicate feature models with different sizes and ECRs. Rows indicate the completion of failure due to memory overflow to build the BDD. Both heuristics performed well for feature models with 1000 features and up to 30% ECR. While Pre-CL-MinSpan failed in only 2% of the cases (1 model), Pre-CL-Size observed memory overflows in 28% of the cases (14 models). For features models with 2000 features, we observed that the number of failures grew proportionally to the increase of the ECR. Pre-CL-MinSpan handled a 10% ECR without a single failure but struggled with ECRs of 20% and higher (54% and 80% failures). Pre-CL-Size was only effective in 64% of the cases for ECRs of 10% or less. None of the heuristics were able to

generate BDDs for feature models with 5000 features for the test cases provided.

Sifting. Space was not an issue for Pre-CL heuristics to generate BDDs for feature models containing up to 1000 features. Hence, we only considered using sifting for larger models. A collection with 50 feature models of 2000 features each and ECR of 20% (the same used in the scalability tests) was considered. Pre-CL heuristics produced the initial variable orders and the only change made was to the experiment configuration was the enabling of sifting in the JavaBDD library. The results shown were not encouraging. Even though memory overflows were prevented successfully none of the 50 test cases completed after 1 hour of processing. Recall that Pre-CL-MinSpan was still able to generate BDDs for 46% of the models (average generation time of about 35 seconds). We observed many calls to the sifting algorithm during the BDD building process each taking several minutes to complete. Therefore, we do not see any real benefits of using sifting to build BDDs for feature models.

8. Conclusion

We have discussed the importance of BDDs to support efficient automated analysis of feature models. We argued that because BDDs are very sensitive to the orderings of its variables it is critical to learn how to order BDD variables in the domain of interest, in our case, the feature modeling domain. We reviewed dynamic and static heuristics including those applied in the domain of logic circuits. Several issues related to ordering BDD variables for feature models were addressed and two new heuristics introduced. We showed experimentally that the heuristics produce high quality variable orders that enable the compilation of large feature models with up to 2,000 features, which was not possible with the previously known heuristics.

References

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proc. of the 13th ACM Great Lakes symposium on VLSI*, 2003.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE*, 2006.

[3] H. R. Andersen. *Binary Decision Diagrams*. Technical University of Denmark, 1997. Lecture notes for 49285, Advanced Algorithms, E97.

[4] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 2006.

[5] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[6] D. Beuche and M. Dalgarno. Software product line engineering with feature models. In *Software Acumen*, 2006. `http://www.methodsandtools.com/PDF/mt200604.pdf`.

[7] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-Complete. *IEEE Transac. on Computers*, 1996.

[8] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN. In *In Int. Symposium on Circuits and Systems*, 1985.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

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

[11] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.

[12] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constriants. In *GPCE*, 2006.

[13] K. Czarnecki and A. Wąsowski. Feature models and logics: There and back again. In *SPLC 2007*. IEEE Press.

[14] K. Czarnecki et al. Generative programming for embedded software: An industrial experience report. In *GPCE*, 2002.

[15] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvement of boolean comparison method based on binary decision diagrams. In *ICCAD*, 1988.

[16] T. Hadzic, R. Jensen, and H. R. Andersen. Notes on calculating valid domains. Manuscript online `http://www.itu.dk/~tarik/cvd/cvd.pdf`, 2006.

[17] T. Hadzic et al. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO Conference*, 2004.

[18] T. Hadzic et al. Calculating valid domains for BDD-based interactive configuration. *CoRR*, abs/0704.1394, 2007.

[19] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[20] S. Q. Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, Dept. of ECE, University of Waterloo, Canada, 2006.

[21] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using BDDs in a logic synthesis environment. In *ICCAD*, 1988.

[22] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[23] M. Mendonca. Efficient compilation techniques for large scale feature models, 2008. `http://csg.uwaterloo.ca/~marcilio/fmcompilation/index.html`.

[24] M. Mendonca, T. T. Bartolomei, and D. Cowan. Decision-making coordination in collaborative product configuration. In *ACM 23rd Symposium on Applied Computing (SAC'08)*, 2008.

[25] J. Moller, H. R. Andersen, and H. Hulgaard. Product configuration over the internet. `http://citeseer.ist.`

psu.edu/531891.html.

[26] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, 1993.

[27] J. Whaley. The JavaBDD library, 2003–2008. http://javabdd.sourceforge.net/.