Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening

Jules White, Brian Dougherty, and Douglas C. Schmidt

Department of Electrical Engineering and Computer Science,
Vanderbilt University,
Nashville, TN, USA
{*jules,briand,schmidt*}*@dre.vanderbilt.edu*

Abstract

Software Product-lines (SPLs) are software architectures that use modular software components that can be reconfigured into different variants for different requirements sets. Feature modeling is a common method used to capture the configuration rules for an SPL architecture. A key challenge developers face when maintaining an SPL is determining how to select a set of architectural features for an SPL variant that simultaneously satisfy a series of resource constraints. This paper presents an approximation technique for selecting highly optimal architectural feature sets while adhering to resource limits. The paper provides the following contributions to configuring SPL architecture variants: (1) we provide a polynomial time approximation algorithm for selecting a highly optimal set of architectural features that adheres to a set of resource constraints, (2) we show how this algorithm can incorporate complex architectural configuration constraints; and (3) we present empirical results showing that the approximation algorithm can be used to derive architectural feature sets that are more than 90%+ optimal.

1 Introduction

Software Product-Lines (SPLs) are a technique for creating reconfigurable software architectures that can be adapted for new requirement sets. For example, an SPL for a face recognition system to identify known cheaters in a casino, as shown in Figure 1, can provide a number of different face recognition algorithms that can be configured depending on the desired accuracy, system cost, and processing power of the hosting infrastructure. Customers with smaller budgets can choose cheaper variants of the SPL that employ less accurate algorithms capable of running effectively on commodity hardware. For more expen-

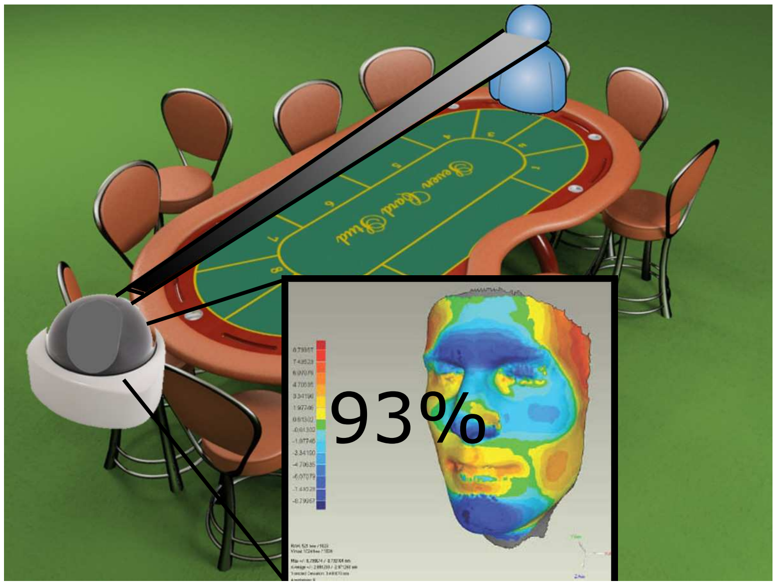

Fig. 1. A Face Recognition System to Identify Cheaters in a Casino

sive variants, algorithms with greater accuracy and correspondingly increased resource consumption can be paired with more expensive custom hardware.

A core part of building an SPL architecture is documenting the rules governing the configuration of the SPL's constituent architectural components. For example, although running two face recognition algorithms in parallel might produce the highest accuracy, a system may not be capable of simultaneously employing two different face recognition algorithms. It is therefore crucial to capture these constraints that guide the configuration of the architecture. *Feature modeling* (Kang et al., 1998), is a commonly used technique to specify an SPL's configuration rules.

Feature models describe an SPL using a tree structure (*e.g.*, shown in Figure 2) where each node in the tree represents a point of architectural variation or increment of functionality. The feature model for an SPL provides a compact representation of all possible architectural variants of the application. Each unique configuration of the SPL architecture—called a *variant*—is described as a set of selected architectural features. Any given architectural feature selection can be validated against its underlying feature model to check if it represents a valid configuration of the SPL.

Choosing the correct set of architectural features for an application is hard because even small numbers of design variables (*i.e.*, small feature sets) can produce an exponential number of design permutations. For example, the relatively simple feature model shown in Figure 3, contains 30 features that can be combined into 300 different distinct architectures. Requirement specifications often try to meet certain goals, such as maximizing face recognition accuracy, that further complicates architectural feature choices.

Resource constraints, such as the maximum available memory or total budget for a system, also add significant complexity to the architectural design process. As shown in Section 4, finding an optimal architectural variant that adheres to both the feature model constraints and a system's resource constraints is an NP-hard problem (Cormen et al., 1990). The manual processes commonly used to select architectural feature sets scale poorly for NP-hard problems.

For large-scale systems—or in domains where optimization is critical—algorithmic techniques are needed to help product-line engineers make informed architectural feature selections. For example, developers can choose the features that are deemed critical for the system or driven by physical concerns that are hard to quantify (such as camera types and their arrangement). An algorithmic technique can then be used to make the remaining architectural feature selections that maximize accuracy while not exceeding the remaining budgetary allocation. Moreover, developers may want to evaluate tradeoffs in architectures, *e.g.*, use a specific camera setup that minimizes memory consumption as opposed to maximizing accuracy.

Existing algorithmic techniques for aiding developers in the selection of architectural variants rely on exact methods, such as integer programming, that exhibit exponential time complexity and poor scalability. Since industrial-size architectural feature models can contain thousands of features, these exact techniques are impractical for providing algorithmic architectural design guidance, such as automated architectural feature selection optimization. With existing techniques, automated feature selection can take hours, days, or longer depending on the problem size. For large problem sizes, this slow solving time makes it hard for developers to evaluate highly optimized design variations rapidly.

This paper presents a polynomial time approximation algorithm, called *Filtered Cartesian Flattening*, that can be used to derive an optimal architectural variant subject to resource constraints. Using Filtered Cartesian Flattening, developers can quickly derive and evaluate different architectural variants that both optimize varying system capabilities and honor resource limitations. Moreover, each architectural variant can be derived in seconds as opposed to the days, hours, or longer that would be required with an exact technique, thereby allowing the evaluation of more architectural variants in a shorter time frame.

This paper provides the following contributions to the study of applying the Filtered Cartesian Flattening algorithm to assist developers in selecting SPL architectural variants:

(1) We prove that optimally selecting architectural feature sets that adhere

to resource constraints is an NP-hard problem.

(2) We present a polynomial time approximation algorithm for optimizing the selection of architectural variants subject to resource constraints.
(3) We show how any arbitrary Multi-dimensional Multiple-choice Knapsack (MMKP) algorithm (MOSER et al., 1997; Pisinger, 1995; Sinha and Zoltners, 1979) can be used as the final step in Filtered Cartesian Flattening, which allows for fine-grained control of tradeoffs between solution optimality and solving speed.
(4) We present empirical results from experiments performed on over 500,000 feature model instances that show how Filtered Cartesian Flattening averages 92.56%+ optimality on feature models with 1,000 to 10,000 features.
(5) We provide metrics that can be used to examine an architectural feature selection problem instance and determine if Filtered Cartesian Flattening should be applied.

The remainder of this paper is organized as follows: Section 2 provides a brief overview of feature modeling; Section 3 presents a motivating example used throught the paper; Section 4 describes the challenges of optimally selecting a set of architectural features subject to a set of resource constraints; Section 5 presents the Filtered Cartesian Flattening approximation algorithm for optimally selecting architectural feature sets; Section 6 presents empirical results showing that our algorithm averages more than 90%+ optimality on feature models ranging from 1,000 to 10,000 features; Section 7 compares our work to related research; and Section 8 presents concluding remarks.

2 Overview of Feature Modeling

Feature modeling (Kang et al., 1998) is a modeling technique that describes the variability in an SPL architecture with a set of architectural features arranged in a tree structure. Each architectural feature represents an increment in functionality or variation in the product architecture. For example, Figure 2 shows a feature model describing the algorithmic variability in a system for identifying faces (Phillips et al., 2000) in images. Each box represents a feature. For example, Linear Discriminant Analysis (LDA) is an algorithm (Parker and Parker, 1996) for recognizing faces in images that is captured in the `LDA` feature.

A feature can (1) capture high-level variability, such as variations in end-user functionality, or (2) document low-level variabilities, such as *software variability* (*e.g.*, variations in software implementation)(Metzger et al., 2007). Each complete architectural variant of the SPL is described as a set of selected features. For example, the feature model in Figure 2 shows how the fea-

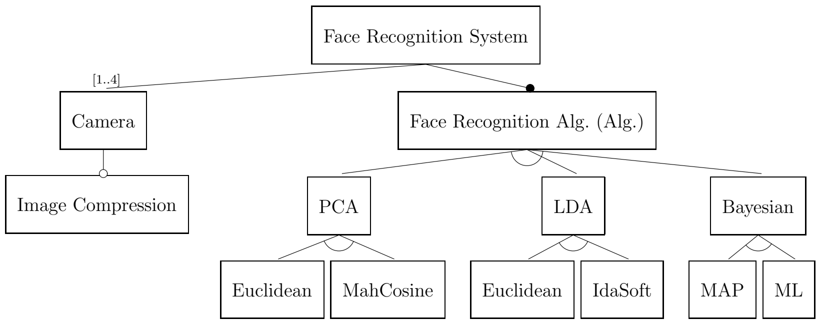

Fig. 2. Example Feature Model

ture set [`Face Recognition System`, `Camera`, `Face Recognition Alg`, `PCA`, `MahCosine`] would constitute a complete and correct feature selection.

The constraints on what constitutes a valid feature selection are specified by the parent child relationships in the tree. Every correct feature selection must include the root feature of the tree. Moreover, if a feature is selected, the feature's parent must also be selected. A feature can have required sub-features indicating refinements to the feature. For example, `Face Recognition System` has a required sub-feature called `Face Recognition Alg.` that must also be selected if `Face Recognition System` is selected. The required relationship is denoted by the filled oval above `Face Recognition Alg.`.

The parent child relationships can indicate variation points in the SPL architecture. For example, `LDA` requires the selection of either of its `Euclidean` or `IdaSoft` sub-features, but not both. The `Euclidean` and `IdaSoft` features form an exclusive-or subgroup, called an *XOR group*, of the Linear Discriminant Analysis (`LDA`) feature that allows the selection of only one of the two children. The exclusive-or is denoted with the arc crossing over the connections between `Euclidean`, `IdaSoft`, and their parent feature. Child features may also participate in a *Cardinality group*, where any correct selection of the child features must satisfy a cardinality expression.

Feature models can also specify a cardinality on the selection of a sub-feature. For example, at least 1 and at most 4 instances of the `Camera` feature must be selected. An unfilled oval above a feature indicates a completely optional sub-feature. For example, a camera can optionally employ `Image Compression`. Finally, a feature can refer to another feature that it requires or excludes that is not a direct parent or child. These constraints are called `cross-tree constraints`.

3 Motivating Example

A key need with SPL architectures is determining how to select a good set of architectural features for a requirement set. For example, given a face recognition system that includes a variety of potential camera types, face recognition algorithms, image formats, and camera zoom capabilities, what is the most accurate possible system that can be constructed with a given budget? The challenge is that with hundreds or thousands of architectural features—and a vastly larger number of architectural permutations—it is hard to analyze the resource consumption and accuracy tradeoffs between different feature selections to find an optimal architectural variant.

3.1 Motivating Example

As a motivating example of the complexity of determining the best set of architectural features for a requirement set, we provide a more detailed example of the face recognition system for identifying known cheaters in a casino. A small example feature model of the face recognition system's architectural features is shown in Figure 3. The system can leverage a variety of algorithms ranging

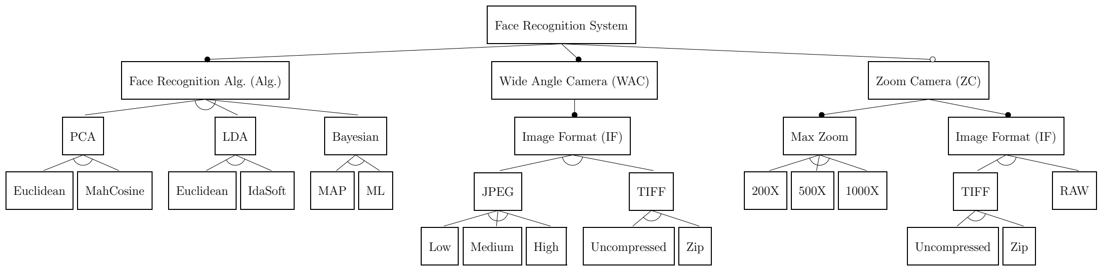

Fig. 3. Face Recognition System Arch. Feature Model

from versions of Linear Discriminant Analysis (LDA) to Bayesian networks. The system requires a wide angle camera, but can be supplemented with a zoom camera to provide closer images of specific faces in the environment. Each camera can produce images in a variety of image formats ranging from lossy low quality JPEG images to lossless RAW images from the camera's CCD sensor.

Each variability point in the architecture, such as the type of face recognition algorithm, affects the overall accuracy and resource consumption of the system. For example, when higher resolution images are obtained by a camera, the overall accuracy of the system can improve. Higher resolution images, however, consume more memory and require more CPU time to process. Depending on the overall system requirements, therefore, choosing higher resolution images to improve accuracy may or may not be possible, depending on the available memory and the memory consumed by other features.

Table 1 captures example information on the accuracy provided—and resources consumed—by some of the architectural features. Each feature is identified by the path through the feature model to reach the feature. For example, the high resolution JPEGs feature is identified by `WAC/IF/JPEG/High`. The choice of architectural features is governed by the overall goal of the system. In this example, we want to maximize face recognition accuracy without exceeding the available memory, CPU, development budget, or development staff. Our architectural goal and resource limits are shown in Table 2.

Arch. Feature	Accuracy	CPU	Memory	Cost	Devel. Staff
WAC/IF/JPEG/High	0.10	8	1024	2	0
WAC/IF/JPEG/Low	0.03	2	128	2	0
...					
ZC/IF/TIFF/Zip	0.13	16	256	30	1
...					
Alg/LDA/Euclidean	0.85	112	2048	300	1
Alg/LDA/IdaSoft	0.84	97	1024	120	0

Table 1
Software Feature Resource Consumption, Cost, and Accuracy

Table 2 lists the architectural resource constraints and goal for the design of the system. The first column lists the goal, which is to maximize the accuracy of the system. Each subsequent column lists a resource, such as total system memory, and the amount of that resource that is available for an architectural variant's features to consume.

Accuracy	CPU	Memory	Cost	Devel. Staff
Maximize	$\leq$ 114	$\leq$ 4096	$\leq$ 330	$\leq$ 1

Table 2
Example Architectural Requirements: Maximize Accuracy Subject to Resource Constraints

4 Challenges of Feature Selection Problems with Resource Constraints

To make well-informed architectural decisions, developers need the ability to easily create and evaluate different architecture variations tuned to maximize or minimize specific system capabilities, such as minimizing total cost or required memory. Generating and evaluating a range of architectures allows developers to gain insights into not only what architectural variants optimize a particular system concern, but also other design aspects, such as patterns that tend to lead to more or less optimal variants. The chief barrier to creating and evaluating a large set of optimized architectural feature models is that generating highly optimized variants is computationally complex and time consuming.

Optimally selecting a set of architectural features subject to a set of resource constraints is challenging because it is an NP-hard problem. To help understand why optimal feature selection problems with resource constraints is NP-hard, we first need to formally define these problems. An architectural feature selection problem with resource constraints is a five-tuple composed of a set of features (F), a set of dependency constraints on the features (C) defined by the arcs in the feature model graph, a function (Fr(i,j))that computes the amount of the j_{th} resource consumed by the i_{th} feature, a set of values or benefits associated with each feature (Fv), and a list of the resource limits for the system (R):

$$P =< F, C, Fr(i,j), Fv, R >$$

The features (F) correspond to the the feature nodes in the feature model graph shown in Figure 3, such as `Bayesian` and `LDA`. The dependency constraints (C) correspond to the arcs connecting the feature nodes, such as `Face Recognition Alg` is a required sub-feature of `Facial Recognition System`. The resource consumption function (Fr) corresponds to the values in columns 3-6 of Table 1, such as the amount of memory consumed by each feature. The feature values set (Fv) corresponds to the accuracy column in Table 1. Finally, the resource limits set (R) corresponds to the resource limits captured in columns 2-4 of Table 2.

We define the solution space to a feature selection problem with resource constraints as a set of binary strings (S) where for any binary string ($s \subset S$) the i_{th} position is 1 if the i_{th} feature in F is selected and 0 otherwise. The subset of these solutions that are valid ($Vs \subset S$) is the set of solutions that satisfy all of the feature model constraints (1) and adhere to the resource limits (2):

$$
\begin{aligned}
Vs = \{s \subset S | \\
& s \rightarrow C, && (1) \\
& \forall j \subset R, (\textstyle\sum_{i=0}^{n} s_i * Fr(i,j)) \leq R_j\} && (2)
\end{aligned}
$$

To prove that optimally selecting a set of architectural features subject to resource constraints is NP-hard, we show below how any instance of an NP-complete problem, the Multi-dimensional Multiple-choice Knapsack Problem (MMKP), can be reduced to an instance of this definition of the optimal feature selection problem with resource constraints.

A traditional knapsack problem is defined as a set of items with varying sizes and values that we would like to put into a knapsack of limited size. The goal is to choose the optimal set of items that fits into the knapsack while maximizing the sum of the items' values. An MMKP problem is a variation on the traditional knapsack problem where the items are divided into sets and at most one item from each set must be placed into the knapsack. The goal remains the same, *i.e.*, to maximize the sum of the items' values in the knapsack.

We provide a simple example of transforming an MMKP problem into a feature selection problem with resource constraints. Figure 4 shows a simple MMKP problem with six items divided into two sets. At most one one of the items A,

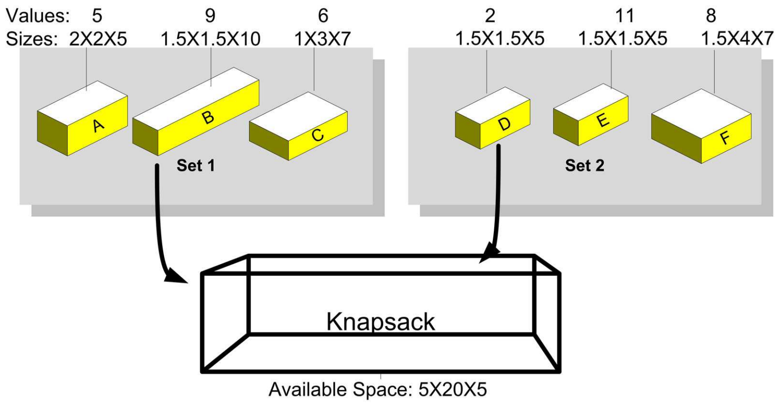

Fig. 4. A Multi-dimensional Multiple-choice Knapsack Problem

B, and C can be in the knapsack at a given time. Moreover, at most one of the items D, E, and F can be in the sack.

To transform the MMKP problem into a feature selection problem with resource constraints, we create a feature model to represent the possible solutions to the MMKP problem, as shown in Figure 5. The generalized algorithm for converting an instance of an MMKP problem into an equivalent feature selection problem with resource constraints is as follows:

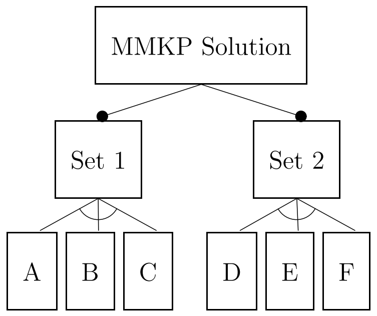

Fig. 5. A Feature Model of an MMKP Problem Instance

(1) Create a root feature denoting the MMKP solution,
(2) For each set, create a mandatory sub-feature of the root feature,
(3) For each set, add an XOR group of sub-features corresponding to the items in the set,
(4) For each item, initialize its feature's resource consumption value entries in the feature properties table to the length, width, and height of the item,
(5) For each item, initialize its feature's value entry in the feature properties table, shown in Table 3, to the item's value, and
(6) Set the total available resources to be the length, width, and height of the knapsack.

Feature	Value	Resource 1 (length)	Resource 2 (width)	Resource 3 (height)
A	5	2	2	5
B	9	1.5	1.5	10
C	6	1	3	7
D	2	1.5	1.5	5
E	11	1.5	1.5	5
F	8	1.5	4	7

Table 3
MMKP Feature Properties Table

Steps 1&2 define the sets (F) and (C) for our feature selection problem. Step 3 creates a table, shown in Table 3, that can be used to define the function $(Fr(i, j))$ to calculate the amount of each resource consumed by a feature. Step 4 initializes the set of values (Fv) defining the value associated with selecting a feature. Finally, Step 5 creates the set of available resources (R).

With this generalized algorithm, we can translate any instance of an MMKP problem into an equivalent feature selection problem with resource constraints. Since any instance of an MMKP problem can be reduced to an equivalent feature selection problem with resource constraints, then feature selection problems with resource constraints must be NP-hard. Any exact algorithm for solving feature selection with resource constraints will thus have exponential time complexity.

5 Filtered Cartesian Flattening

This section presents the Filtered Cartesian Flattening (Filtered Cartesian Flattening) approximation technique for optimal feature selection subject to resource constraints. Filtered Cartesian Flattening transforms an optimal feature selection problem with resource constraints into an approximately equivalent MMKP problem, which is then solved using an MMKP approximation algorithm. The MMKP problem is designed such that any correct answer to the MMKP problem is also a correct solution to the feature selection problem (but not necessarily vice-versa). Filtered Cartesian Flattening allows developers to generate highly optimal architectural variants algorithmically in polynomial-time (roughly ∼10s for 10,000 features), rather than in the exponential time of exact algorithmic techniques, such as integer programming.

As shown below, Filtered Cartesian Flattening addresses the main challenge from Section 4, *i.e.*, the difficulty of selecting a highly optimal feature selection in a short amount of time. The key to Filtered Cartesian Flattening's short solving times is that it is a polynomial time approximation algorithm that trades off some solution optimality for solving speed and scalability.

The Filtered Cartesian Flattening algorithm, which we will describe in the following subsections, is listed in the APPENDIX.

5.1 Step 1: Cutting the Feature Model Graph

The first step in Filtered Cartesian Flattening, detailed in code listing (2) of the APPENDIX, is to begin the process of producing a number of independent MMKP sets. We define a choice point as a place in an architectural feature model where a configuration decision must be made (*e*.g., XOR Group, Optional Feature, etc.). A choice point, A, is independent of another choice point, B, if the value chosen for choice point A does not affect the value chosen for choice point B. An MMKP problem must be stated so that the choice of an item from one set does not affect the choice of item in another set.

For example, the choice point containing `Image Compression` in Figure 2 is independent of the choice point containing `MAP` and `ML`, *i.e.*, whether or not image compression is enabled does not affect the type of `Bayesian` algorithm chosen. The choice point of the type of face recognition algorithm, which contains the feature `Bayesian`, is not independent of the choice point for the type of Bayesian algorithm (*e.g.*, the XOR group with `MAP` and `ML`).

Filtered Cartesian Flattening groups choice points into sets that must be independent. Each group will eventually produce one MMKP set. Starting from the root, a depth-first search is performed to find each optional feature that has no ancestors that are choice points. A cut is performed at each of these optional features with no choice point ancestors to produce a new independent sub-tree, as shown in Figure 6. After these cuts are made, if the sub-trees have cross-tree constraints, they may not yet be completely independent. These cross-tree constraints are eliminated in Step 4, described in Section 5.4

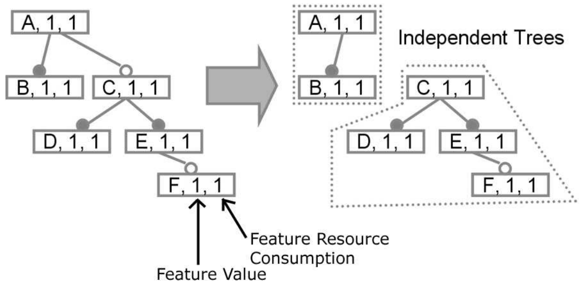

Fig. 6. Cutting to Create Independent Sub-trees

5.2 Step 2: Converting to XOR

Each MMKP set forms an XOR group of elements. Since MMKP does not support any other relationship operators, such as cardinality, we must convert the configuration solution space captured in each feature model sub-tree into an equivalent representation as a series of partial configurations related through XOR. Since a feature model allows hierarchical modeling and cardinality constraints, the conversion to XOR can require an exponential number of partial configurations for the XOR representation.[1] The filtering process of Filtered Cartesian Flattening is an approximation step that puts a polynomial bound on the number of configuration permutations that are encoded into the XOR representation to avoid this state explosion.

The first step in converting to XOR is to convert all Cardinality groups and optional features into XOR groups. Cardinality groups are converted to XOR by replacing the cardinality group with an XOR group containing all possible combinations of the cardinality group's elements that satisfy the cardinality expression. Since this conversion could create an exponential number of elements, we bound the maximum number of elements that are generated to a

[1] This state explosion is similar to what happens when a State Chart with hierarchy is converted to its equivalent Finite State Machine representation Harel (1987).

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 thrown away.

Any number of potential filtering options can be used. Our experiments in Section 6 evaluated a number of filtering strategies, such as choosing the K highest valued items, a random group of K items, and a group of K items evenly distributed across the items's range of resource consumptions. The best results occurred when 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 partial configuration. This sorting criteria has been used successfully by other MMKP algorithms (Akbar et al., 2001). An example conversion with $K = 3$ and random selection of items is shown in Figure 7.

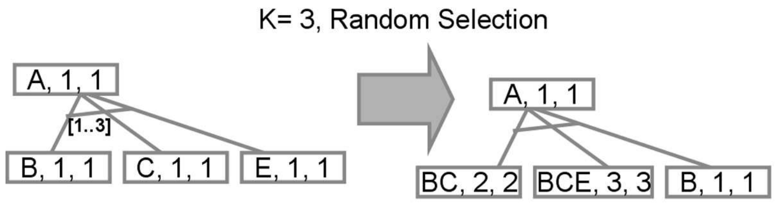

Fig. 7. Converting a Cardinality Group to an XOR Group with K=3 and Random Selection

Individual features with cardinality expressions attached them are converted to XOR 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 8.

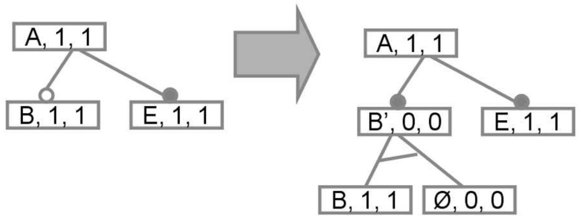

Fig. 8. Converting an Optional Feature into an XOR Group

5.3 Step 3: Flattening with Filtered Cartesian Products

For each independent sub-tree of features that now only have XOR and required relationships, an MMKP set needs to be produced. Each MMKP set needs to consist of a number of partial configurations that could be produced from each sub-tree. To create the partial configurations that constitute each MMKP set, we perform a series of recursive flattening steps using filtered Cartesian products, as shown in code listing (4) in the APPENDIX.

The procedure `flatten` takes a feature and recursively flattens its children into a MMKP set that is returned as a list. The list is constructed such that each item represents a complete and correct configuration of the feature and its descendants. The first step in the algorithm (5) simply takes a feature with no children and returns a list containing that feature, *i.e.*, if the feature's subtree contains only a single feature, the only valid configuration of that subtree is the single feature. The second step (6) merges the valid partial configurations of two nested XOR groups into a single partial configuration by merging their respective partial configuration sets into a single set. A visualization of this step is shown in Figure 9.

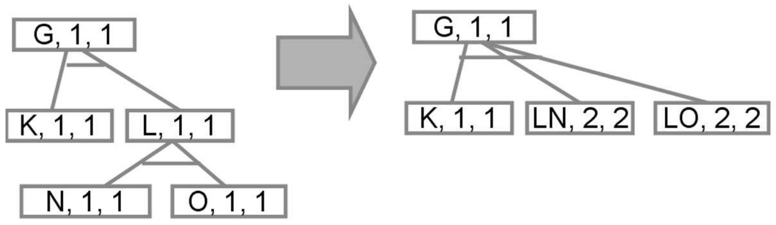

Fig. 9. Flattening an XOR Group

The third step (7) takes all required children of a feature and produces a partial configuration containing a filtered Cartesian product of the feature's children, *i.e.*, the step selects a finite number of the valid configurations from the set of all possible permutations of the child features' configurations. A visualization of this step is shown in Figure 10. In code listing (8) in the

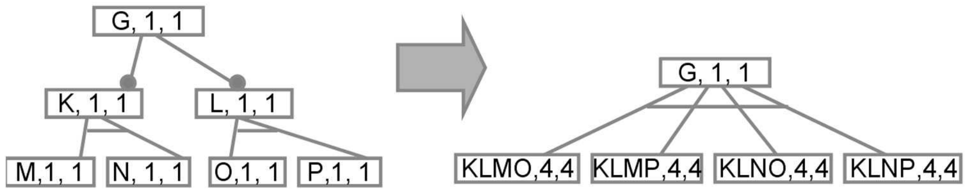

Fig. 10. A Cartesian Product of Required Children

APPENDIX, the Cartesian product is filtered identically to the way filters are used in Section 5.2. The filter chooses K elements from the Cartesian product of the two sets using a selection strategy. The experiments in Section 6 shows that a value of K=400 produced a good blend of speed and optimality.

Once each independent sub-tree has been converted into a set of partial configurations, we must mark those sets that 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. Either a partial configuration from the set is selected or $\emptyset$ (representing no selection) is chosen. This method is a standard MMKP technique for handling situations where choosing an item from some sets is optional. Since the root feature must always be chosen, a partial configuration from its sub-tree's set must also be chosen, so the $\emptyset$ item is not added to its set.

5.4 Step 4: Handling Cross-tree Constraints

If any of the partial configurations in the MMKP sets contain cross-tree constraints, these constraints must eliminated before the MMKP solver is used. There are two cases for the cross-tree constraints that must be handled:

(1) A partial configuration has a cross-tree constraint that refers to a feature in a sub-tree other than the sub-tree that produced its containing MMKP set.
(2) A partial configuration has a cross-tree constraint that refers to a feature within the same sub-tree that produced its containing MMKP set.

The first case is handled by applying a series of filtered Cartesian products to each series of two sets that is connected through one or more cross-tree constraints. During the process of calculating the Cartesian product, when two partial configurations are chosen from each of the two sets, the combination of the configurations is validated to ensure that it does not violate any cross-tree exclusionary constraints. If the combination violates a cross-tree excludes constraint, the combined configuration is not added to the filtered Cartesian product of the two sets. In the case that a violation occurs, a constant number of retries, w, can be performed to find an alternate pair of compatible configurations. If no compatible pair is found within w tries, K is decremented for that set, and the Cartesian product continues.

The second case is handled by checking the validity of each partial configuration that contains one or more cross-tree constraints. Each of these partial configurations is checked to ensure that it adheres to its cross-tree constraints. If the configuration is valid, no changes are made. Invalid configurations are removed from their containing MMKP set. Cross-tree constraints within the same sub-tree are always handled after cross-tree constraints between sub-trees have been eliminated.

5.5 Step 5: MMKP Approximation

The first four 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 sub-trees of the feature model.

The final steps in deriving an optimal architectural feature selection involve running an existing MMKP approximation algorithm to select a group of partial configurations to form the architectural feature selection and then to combine these partial configurations into a complete architectural variant. For our implementation of Filtered Cartesian Flattening, we used a simple modification of the Modified Heuristic (M-HEU) algorithm (Akbar et al., 2001) that puts an upper limit on the number of upgrades and downgrades that can be performed. Since Filtered Cartesian Flattening produces an MMKP problem, we can use any other MMKP approximation algorithm, such as the Convex Hull Heuristic algorithm (C-HEU) (Mostofa Akbar et al., 2006), which uses convex hulls to search the solution space. Depending on the algorithm chosen, the solution optimality and solving time will vary.

The items in the MMKP sets are built by concatenating the partial configurations of feature sub-trees during Cartesian products. With this arrangement, architectural feature configuration solutions can readily be extracted from the MMKP solution since they consist of a partial configurations represented as a series of strings containing the labels of features that should be selected.

5.6 Algorithmic Complexity

The algorithmic complexity of Filtered Cartesian Flattening's constituent steps can be decomposed as follows (where n is the number of features):

- The first step in the Filtered Cartesian Flattening algorithm—cutting the tree—requires O(n) time to traverse the tree and find the top-level optional features where cuts can be made.
- 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, add no additional 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). Our experiments in Section 6 selected the K items with the best value to resource consumption ratio. With this strategy, the sets must be sorted, yielding O($Kn * n \log n$).

- The third step in the algorithm requires flattening at most n groups using filtered Cartesian products, which yields a total time of $\mathrm{O}(Kn * S)$.
- The fourth step in the algorithm requires producing filtered Cartesian products from at most n sets with w retries. Each configuration can be checked in $\mathrm{O}(c \log n)$, where c is the maximum number of cross-tree constraints in the feature model. The total time to eliminate any cross-tree constraints between sets is $\mathrm{O}(wKn * S * c \log n)$. The final elimination of invalid configurations within individual sets requires $\mathrm{O}(cn \log n)$, yielding a total time of $\mathrm{O}(wKn * S * c \log n + cn \log n)$
- The solving step incurs the algorithmic complexity of the MMKP approximation algorithm chosen. With M-HEU, the algorithmic complexity is $\mathrm{O}(mn^2(l-1)^2)$, where m is the number of resource types, n is the number of sets, and l is maximum items per set.
- The final step, extracting the feature selection, can be performed in $\mathrm{O}(n)$ time.

This analysis yields a total general algorithmic complexity of $\mathrm{O}(n+(Kn*S)+(Kn*S)+(wKn*S)+MMKP+n) = \mathrm{O}(wKn*S*c \log n+cn \log n+MMKP)$. If there are no cross-tree constraints, the complexity is reduced to $\mathrm{O}(Kn * S + MMKP)$. Both algorithmic complexities are polynomial, which means that Filtered Cartesian Flattening scales significantly better than exponential exact algorithms. The results in Section 6.2 show that this translates into a significant decrease in running time compared to an exact algorithm.

5.7 Technique Benefits

Beyond the benefit of providing polynomial-time approximation for optimal feature selection problems with resource constraints, Filtered Cartesian Flattening exhibits the following other desirable properties:

One-time Conversion to MMKP: The Filtered Cartesian Flattening flattening process to create an MMKP problem need only be performed once per feature model. As long as the structure and resource consumption characteristics of the features do not change, the same MMKP problem representation can be used even when the resource allocations (we merely update the knapsack size) or desired system property to maximize change.

Flexible Filtering and Solving Strategies: Due to the speed of the Filtered Cartesian Flattening process, a number of different filtering strategies can be used and each resultant MMKP problem stored and used for optimization. In fact, to produce the most optimal results, a number of MMKP problems can be produced from each feature model and then each MMKP problem solved with several different MMKP techniques, and the most optimal solution produced can be used. Since there are multiple problem representations and multiple algorithms used to solve the problem, there is a much lower probability that all of the representation/algorithm combinations will produce a solution with low optimality.

Flattening Parallelization: Another desirable property of Filtered Cartesian Flattening is that it is amenable to parallelization during the phase that populates the MMKP sets with partial configurations. After each subtree is identified, the Filtered Cartesian Flattening flattening process for each subtree can be run in parallel on a number of independent processors or processor cores.

Exact MMKP Algorithms Compatiblity: Finally, although we have focused on approximation algorithms for the MMKP phase of Filtered Cartesian Flattening, exact methods, such as integer programming, can be used to solve the MMKP problem. In this hybrid scenario, Filtered Cartesian Flattening would produce an approximate representation of the architectural feature model solution space using an MMKP problem and the exact optimal MMKP answer would be obtained. Filtered Cartesian Flattening allows the use of a wide variety of both Cartesian flattening strategies and MMKP algorithms to tailor solving time and optimality.

6 Results

This section presents empirical results from experiments we performed to evaluate the types of architectural feature selection problem instances on which Filtered Cartesian Flattening performs well and those for which it does not. When using an approximation algorithm, such as Filtered Cartesian Flattening, that does not guarantee an optimal answer a key question is how close the algorithm can get to the optimal answer. Another important consideration is what problem instance characteristics lead to more/less optimal answers from the algorithm. For example, if the algorithm attempts to derive an architectural variant for the face recognition system, will a more optimal variant be found when there is a larger or smaller budget constraint?

We performed the following two sets of experiments to test the capabilities of Filtered Cartesian Flattening:

- **Effects of MMKP problem characteristics.** Since Filtered Cartesian Flattening uses an MMKP approximation algorithm as its final solving step, we first performed experiments to determine which MMKP problem characteristics had the most significant impact on the MMKP approximation algorithm's solution optimality.

- **Effects of feature selection problem characteristics.** Our next set of experiments were designed to test which problem characteristics most influenced the entire Filtered Cartesian Flattening technique's solution optimality. These experiments also included a large experiment that derived Filtered Cartesian Flattening's average and worst optimality on a set of 500,000 feature models.

All experiments used 8 dual processor 2.4ghz Intel Xenon nodes with 2 GB RAM on Vanderbilt University's ISISLab cluster (`www.isislab.vanderbilt.edu`). Each node was loaded with Fedora Core 4. A total of two processes (one per processor) were launched on each machine enabling us to generate and solve 16 optimal feature selection with resource constraints problems in parallel.

6.1 Testing MMKP Problem Characteristics

To determine the extent to which the various attributes of MMKP problems would affect the ability of the solver to generate a highly optimal solution, we generated several MMKP problems with a single parameter adjusted. These problems were then solved using the MMKP approximation algorithm described in Section 5.5. Solutions were rated by their percentage of optimality vs. the optimal solution ($\frac{MMKPApproximationAnswer}{OptimalAnswer}$) (we used the problem generation technique devised by (Akbar et al., 2001) to generate random MMKP problem instances for which we knew the optimum answer). Our test problems included a mix of problems with a correlation between value and total resource consumption and those without any correlation.

MMKP problem instances can vary across a number of major axes. Problem instances can have larger and smaller numbers of sets and items per set. The range of values and resource consumption characteristics across the items can follow different distributions. We examined each of these MMKP problem attributes to determine which ones lead to the generation of solutions with a higher degree of optimality. Each experiment was executed thirty times and averaged to normalize the data.

First, we manipulated the total number of sets in an MMKP problem. The Filtered Cartesian Flattening algorithm produces one set for each independent subtree in the feature model. This experiment allowed us to test how feature models with a large number of independent subtrees and hence a large number of MMKP sets would affect solution optimality. Figure 11 shows that as the total number of sets was increased from 10 to 100, the solution optimality only varied a small amount, staying well above 95% optimal. These results are nearly identical to (Akbar et al., 2001), where the M-HEU MMKP approximation algorithm, which was the basis of our MMKP solver, produced solutions

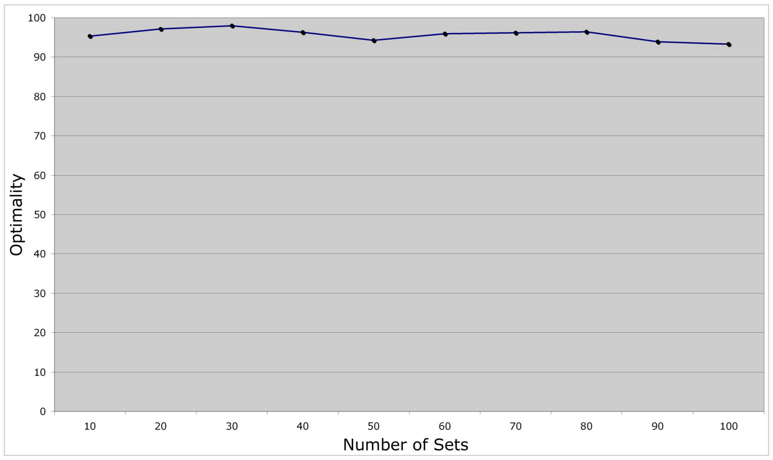

Fig. 11. Total Number of Sets

well above 98% optimal regardless of the number of sets or items per set.

We next varied the number of items in each MMKP set. Figure 12 shows that an increase from 500 to 10,000 items per set has almost no affect the optimality of the solution. Regardless of the number of items per set, the

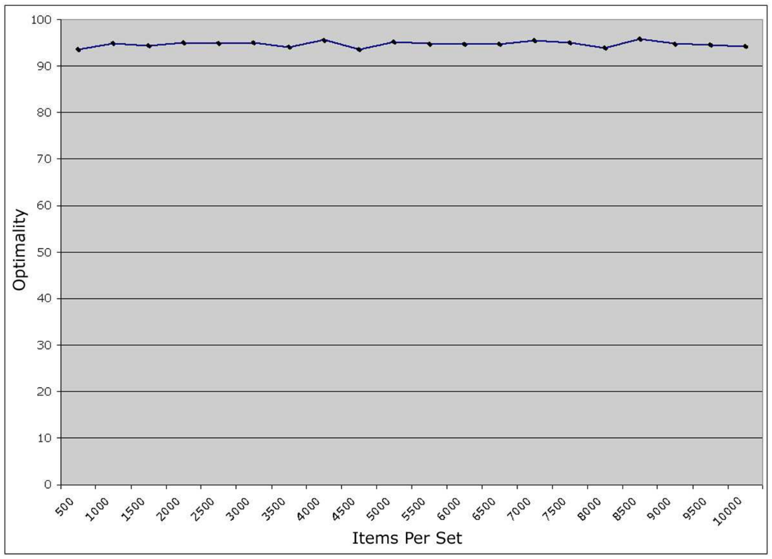

Fig. 12. Items per Set

generated solution was well over 90% optimal. Based on this data, we conclude that the number of sets and total items per set do not significantly impact the optimality of the solution produced by the MMKP solver. This result implies that architectural feature models for very large industrial systems will not be problematic for the MMKP phase of Filtered Cartesian Flattening.

While the items per set and number of sets have little affect on the optimality of a solution, the number of resources, and the amount of resources consumed by items were found to negatively impact the ability of the solver to find a solution with high optimality. Figure 13 shows the affect of raising the minimum amount of resources consumed by an item. The optimality drops drastically as

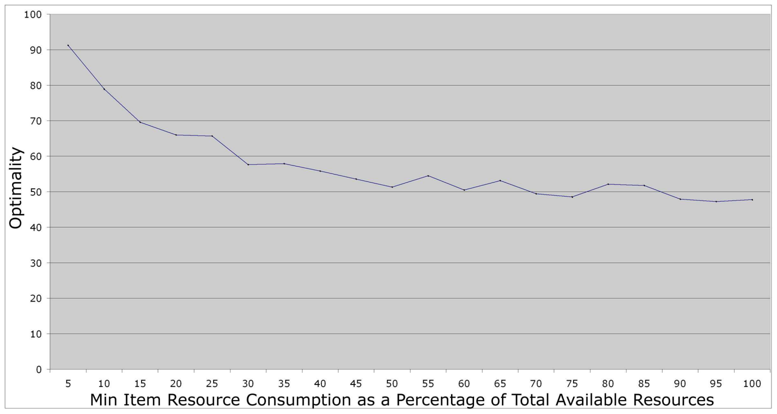

Fig. 13. Minimum Resource Consumption per Item

the minimum amount of resources consumed by an item becomes a larger percentage of the total available resources. For a solution to maintain a forecasted optimality of over 80% percent, the minimum amount of resources consumed by an item must be less than 10% percent of the total amount of available resources. Increasing the minimum amount of resources consumed by an item causes more items to consume a relatively large share of the total available resources.

The results from the experiment that gradually increased the minimum item resource consumption led us to hypothesize that the MMKP solver will produce less optimal solutions when the average item consumes a very large percentage of the available resources. We performed another experiment where we (1) calculated a resource tightness metric that measured the average resource consumption of the items and (2) estimated how many items with the average resource consumption could fit into the available resource allocation, *i.e.*, how many of the average sized items could be expected to fit into the knapsack. Our tightness metric was calculated as:

$$\frac{\sqrt{R_0^2 + \ldots R_m^2}}{\sqrt{(\sum_{i=0}^{n} r(i,0)^2 + \ldots r(i,m)^2)/n}}$$

where m is the total number of resource types, R_i is the maximum available amount of the i_{th} resource, and $r(i,j)$ is the amount of the j_{th} resource consumed by the i_{th} item.

The results from the resource tightness experiment are shown in Figure 14. The x-axis shows the estimated number of average sized items that are expected to fit into the knapsack for a feature model with 50 sets. As shown in the figure, there is a dramatic dropoff in optimality when less than 1.65 average sized items can fit in the knapsack. The exact value for the tightness metric

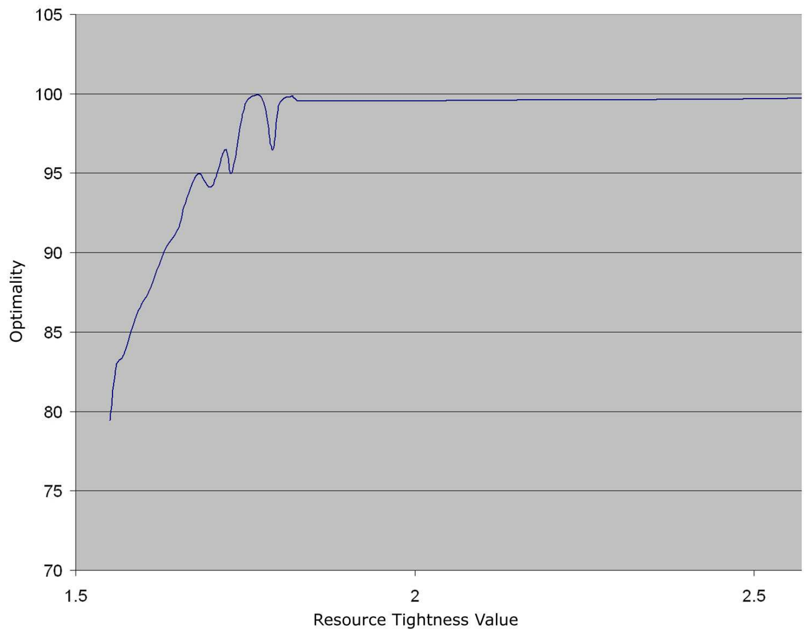

Fig. 14. Effect of Resource Constraint Tightness on MMKP Optimality

at which the dropoff occurs varies based on the number of MMKP sets. With 100 sets, the value was ∼1.83.

The fewer average items that can fit into the knapsack, the more likely the solver is to make a mistake that will fill up the knapsack and widely miss the optimal value. This result implies that the Filtered Cartesian Flattening algorithmic approach works well when making are a relatively large number of finer-grained feature selection decisions. For architectures with a few very coarse-grained decisions, a developer or exact technique (Benavides et al., 2007) is more likely to pick a more appropriate architectural variant.

Resource tightness also played a role in how the total number of resource types affected solution optimality. Figure 15 shows how the optimality of solving problems with 50 sets was affected as the total number of resource types climbed from 2 to 95. For this experiment, the tightness metric was kept above the 1.65 dropoff threshold. As can be seen, the total number of resources had a relatively slight impact of approximately 5% on solution optimality. The results in Figure 16, however, are quite different. In the experiment that produced Figure 16, the tightness metric was kept at a relatively constant 1.55, *i.e.*, below the dropoff value. As shown by the results, the total number of resource types had a significant impact on solution optimality.

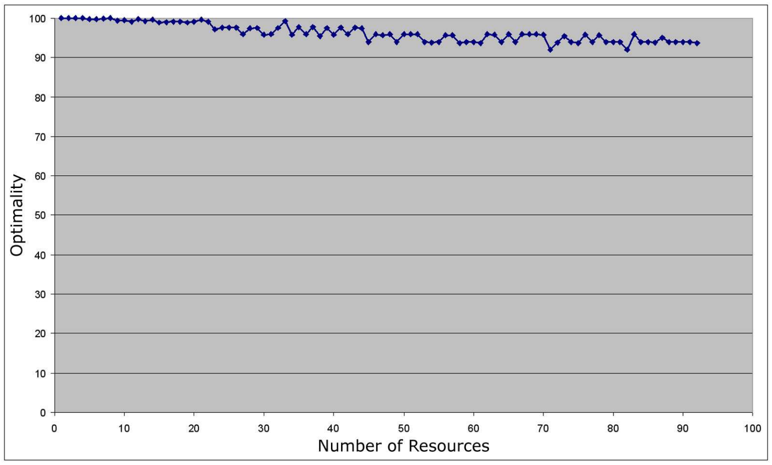

Fig. 15. Total Number of Resources

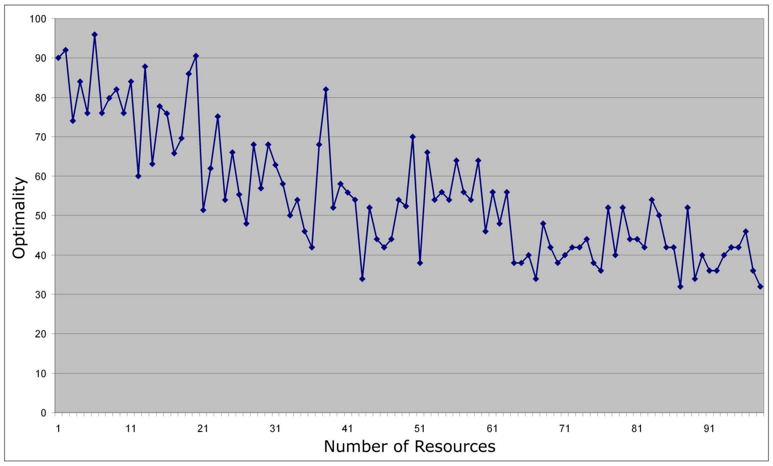

Fig. 16. Total Number of Resources

6.2 Comparing Filtered Cartesian Flattening to CSP-based Feature Selection

Our initial tests with Filtered Cartesian Flattening compared its performance and optimality on small-scale feature selection problems to the Constraint Satisfaction Problem (CSP) based feature selection technique described in (Benavides et al., 2005). This technique uses a general-purpose constraint solver to derive a feature selection. For these small scale- problems, we tracked the time required for Filtered Cartesian Flattening to find a solution vs. the CSP-based technique based on open-source Java Choco constraint solver (`choco-solver.net`). For each solution, we compared Filtered Cartesian Flattening's answer to the guaranteed optimal answer generated by the CSP-based technique.

Figure 17 shows the time required for Filtered Cartesian Flattening and the CSP-based technique to find architectural variants in feature models with varying numbers of XOR groups. The x-axis shows the number of XOR groups

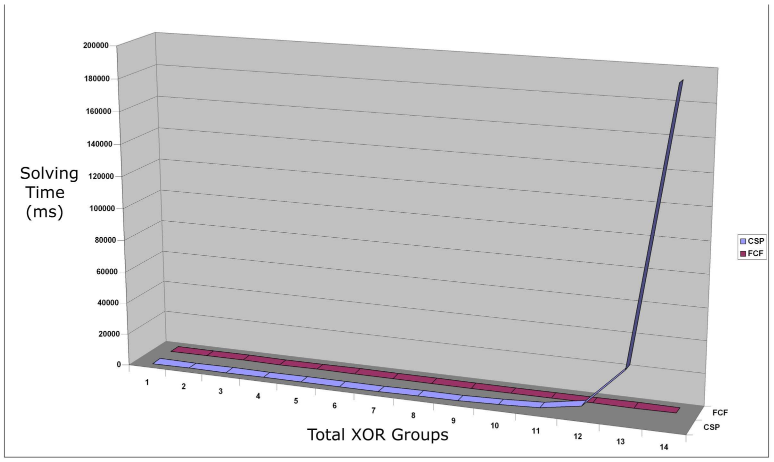

Fig. 17. Comparison of Filtered Cartesian Flattening and CSP-based Feature Selection Solving Times

in the models and the y-axis displays the time required to find an architectural variant. The total features in each model was ∼3-10 times the number of XOR groups (the maximum size was < 140 features). Each feature-model had a maximum of 10% of the features involved in a cross-tree constraint, $c \leq 0.1n$. As shown in the figure, the CSP-based technique initially requires approximately 30ms to find a solution. The CSP technique's time, however, quickly grows at an exponential rate to over 198,000ms. In contrast, Filtered Cartesian Flattening required less than 1ms for every feature model.

Even though Filtered Cartesian Flattening ran substantially faster than the CSP-based technique, it still provided a high level of optimality. Overall, the solutions generated by Filtered Cartesian Flattening were 92% optimal compared to 100% optimal for the CSP-based technique. The Filtered Cartesian Flattening solution with the lowest optimality was 80% optimal. Although Filtered Cartesian Flattening does not provide 100% optimal results, it can be used to derive good architectural variants for architectures that are too large to solve with an exact technique.

6.3 Filtered Cartesian Flattening Test Problem Generation

Due to the exponential time curve required to solve a feature selection problem using an exact technique, it was not possible to solve large-scale problems using both Filtered Cartesian Flattening and an exact technique. This section presents the problem generation technique we used to create large-scale feature selection problems for which we knew the optimal answer. This problem generation approach allowed us to generate extremely large problems with a known optimal solution that were not feasible to solve with an exact technique.

Filtered Cartesian Flattening problem instances vary based on the structural properties of the feature model tree, such as the percentage of XOR groups, max depth, and maximum number of children per feature. The MMKP properties tested in Section 6.1, such as the resource tightness of the problem, can also vary based on how features consume resources. We tested the effect of these problem characteristics by both generating problem instances that exhibited a specific characteristic and by performing post-mortem analysis on the results of solving over 500,000 random Filtered Cartesian Flattening problem instances. The post-mortem analysis determined the problem characteristics associated with the problem instances that were solved with the worst optimality.

To create test data for the Filtered Cartesian Flattening technique, we generated random feature models and then created random feature selection problems with resource constraints from the feature models. For example, we first generated 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 then randomly generated a series of available resource values and ask Filtered Cartesian Flattening 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 Filtered Cartesian Flattening answer to the optimum answer. No models included any cross-tree constraints because there are no known methods for generating large feature selection problems that include cross-tree constraints and have a known optimal solution.

In an effort to make the feature models as representative of real architectural feature models as possible, we created models with a number of specific characteristics. For example, developers with significant object-oriented development experience often create models where commonality is factored into parent features, identical to how an inheritance hierarchy is built. Figure 3 shows a hierarchy used to categorize the various facial recognition algorithms. SPL architectural analysis techniques, such as *Scope, Commonality, Variability Analysis* (Coplien et al., 1998) are used to derive these hierarchies.

Developers desires to provide a well structured hierarchy has two important ramifications for the feature model. First, feature models typically have a relatively limited number of child features for each feature. Hierarchies are used to model a large number of child features as subtrees rather than simply a long list of alternatives. Second, the actual features that consume resources and provide value are most often the leaves of the feature model. In the categorization of facial recognition algorithms shown in Figure 3, the actual resource consumption and accuracy of the algorithm is not specifically known until reaching one of the leaves, such as Euclidean or MahCosine. To mirror these properties of developer-created feature models, we limited the number of child features of a feature to 10 and heavily favored the association of resource consumption and value with the leaves of the feature model.

We used a feature model generation infrastructure that we developed previously (White et al., 2008). A key challenge was determining a way to randomly assign resource consumption values and values to features such that we knew the exact optimum value for the ideal feature selection. Moreover, we needed to ensure that the randomly generated problems would not exhibit characteristics that would make them easily solved by specific MMKP algorithms. For example, if every feature in the optimum feature selection also had the highest value in its selection set, the problem could be solved easily with a greedy algorithm.

To assign resource consumption values to features and generate random available resource allocations, we used a modified version of the algorithm in (Akbar et al., 2001) to ensure that the highest valued features were no more likely part of the optimal solution than any other feature. The steps to generate a feature selection problem with k different resource types and n features were as follows:

(1) Generate a k-dimensional vector, r_a, containing random available allocations for the k resource types,
(2) Randomly generate a slack value v_s,
(3) Randomly generate an optimum value v_{opt},
(4) For each top-level XOR group, q, in each independent sub-tree, randomly choose a feature, f_{qj}, to represent the optimal configuration and assign it value $opt_{qj} = v_{opt}$,
(5) For each optimal feature, assign it a k dimensional resource consumption vector, r_{qj}, such that the sum of the components of the optimal resource consumption vectors exactly equal the available resource allocation vector, $\sum r_{qj} = r_a$,
(6) For each top-level XOR group member f_i that is not the optimal feature f_{qj} in its group either:
    - assign the feature value v_i, where $v_i < (opt_{qj} - v_s)$ and randomly assign it a resource consumption vector
    - assign the feature value v_i, where $opt_{qj} < v_i < opt_{qj} + v_s$, and randomly assign f_i a resource consumption vector such that each component is greater than the corresponding component in r_{qj}. After each XOR group's features are completely initialized, set $v_s = max(v_i) - opt_{qj}$, where $max(v_i)$ is the the highest value of any item in the XOR group.
(7) For each feature in a top-level XOR group, reset the available resources vector to the feature's resource consumption vector, reset the optimum value to the feature's value, and recursively apply the algorithm, treating the feature as the root of a new sub-tree

6.4 Filtered Cartesian Flattening Optimality

After determining the key MMKP problem characteristics that influence the optimality of the MMKP phase of Filtered Cartesian Flattening, we ran a series of experiments to evaluate the parameters that affect the feature model flattening phase. Figure 18 presents results illustrating how the percentage of features involved in XOR groups within the feature model affects solution optimality. As shown in this figure, as the percentage of features in XOR

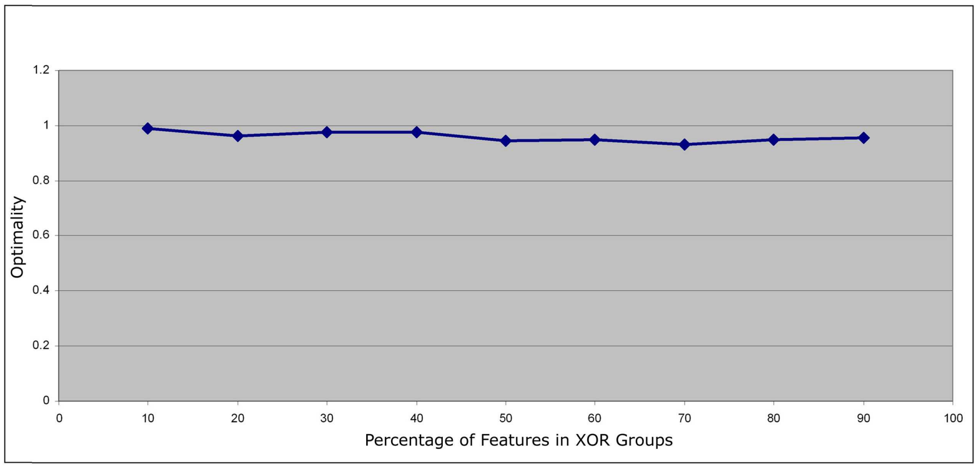

Fig. 18. Effect of Feature Model XOR Percentage on Filtered Cartesian Flattening Optimality

groups increases from 10% to 90% of features, there is a negligible impact on optimality of the solutions produced by Filtered Cartesian Flattening.

We tested a wide range of other Filtered Cartesian Flattening properties, such as the maximum depth and the maximum branching factor of the feature model tree, and saw no impact on solution optimality. Other experiments included tests that assigned and distributed value and resource consumption to sub-trees in correlation to the size of the sub-tree. We also experimented with feature models that evenly distributed value and resource consumption across all features as opposed to clustering resource consumption and value towards the leaves. The effect of different value ranges was also tested.

In each case, we observed no affect on solution optimality. The result graphs from these experiments have been omitted for brevity. As discussed in Section 6.5, our resource tightness metric had the most significant impact on Filtered Cartesian Flattening solution optimality, just as it did with MMKP approximation optimality.

Our largest experiment checked the range of solution optimalities produced by using Filtered Cartesian Flattening to solve 450,000 optimal feature selection problems with resource constraints. The total number of features was set to 1,000, the XOR Group percentage to 50%, $K = 2500$, and the resource tightness metric was greater than 2.0 for the majority of the problem instances (well above the dropoff point). As shown in Figure 19, the results are presented with a histogram showing the number of problem instances that were solved with a given optimality. The overall average optimality across all instances

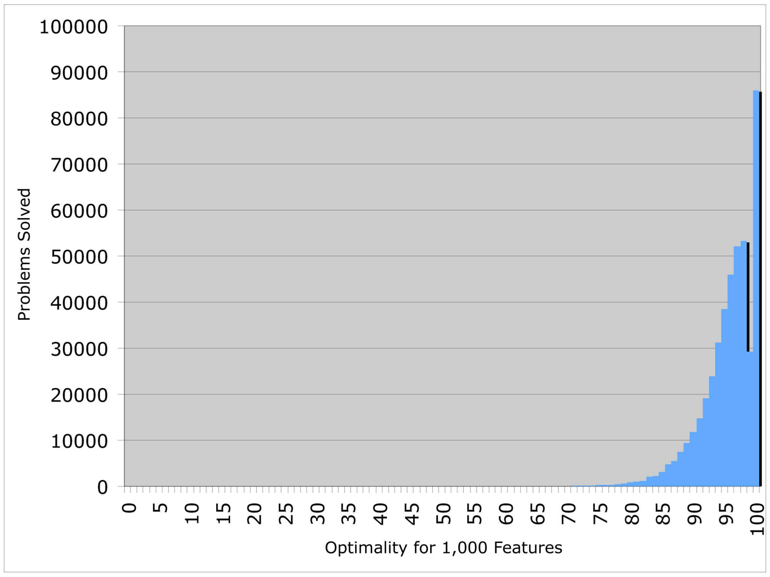

Fig. 19. A Histogram Showing the Number of Problems Solved with a Given Optimality from 450,000 Feature Models with 1,000 Features

was 95.54%. The lowest solution optimality observed was 72%.

Figure 20 presents data from solving approximately 8,000 feature selection problems with 10,000 features. Again, we used a filtering scheme with $K = 2500$ that chose the K items with the best ratio of value to weight. The average optimality across all problem instances was approximately 92.56%.

Across all feature model sizes (both 1,000 and 10,000 features), 90% of the problem instances were solved with an optimality greater than ∼91%. Moreover, 99% were solved with an optimality greater than ∼80%. These result cutoffs only hold when the tightness metric is above the drop-off value.

An interesting result can be seen by comparing Figures 20 and 19. As the number of features increases, the range of solution optimalities becomes much more tightly clustered around the average solution optimality. Akbar's results (Akbar et al., 2001) showed an increase in M-HEU solution optimality as the number of sets and items per set increased. Our results showed a slight decrease of 3% in average solution optimality for Filtered Cartesian Flattening as the total features increased from 1,000 to 10,000. We expect that the slight decrease is a result of more potentially good partial configurations being filtered out during the Filtered Cartesian Flattening Cartesian flattening phase.

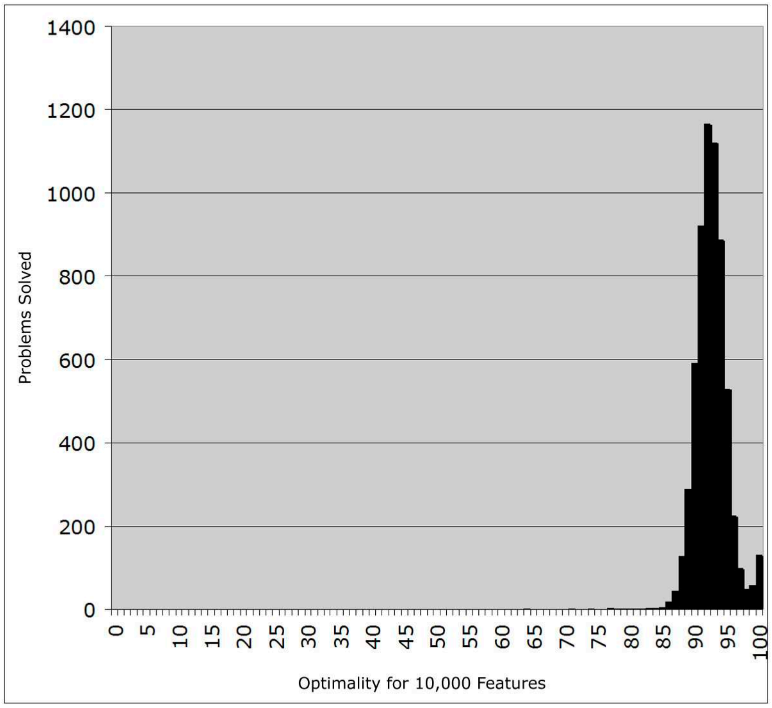

Fig. 20. A Histogram Showing the Number of Problems Solved with a Given Optimality from 8,000 Feature Models with 10,000 features

6.5 Summary and Analysis of Experiment Results

From the data we obtained from our Filtered Cartesian Flattening experiments, we confirmed that the key predictor of MMKP solution optimality—resource tightness—was also applicable to Filtered Cartesian Flattening problems. For all experiments we ran, those problems that were solved with less than 70% optimality had an average resource tightness metric of 0.94, which is well below the dropoff point of roughly 1.65 that we observed for 50 sets. Moreover, the max tightness value for these problems was 1.67, which is right at the edge of the dropoff.

Although a low value for the resource tightness metric indicates that a low optimality is possible, it does not guarantee it. Some problems with tightness metrics below the drop-off were solved with 100 or 90%+ optimality. Once the MMKP problem representation is produced, calculating the tightness metric is an O(n) operation. Due to the ease of calculating the resource tightness metric, developers should always use it to rule out problem instances were Filtered Cartesian Flattening is unlikely to produce an 80-90%+ optimal solution.

7 Related Work

This section describes related work on algorithmic techniques for feature selection and resource allocation and compares it with our Filtered Cartesian Flattening algorithm.

7.1 Exact Techniques

Benavides et al. (Benavides et al., 2005) 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 (White et al., 2007). 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, Filtered Cartesian Flattening performs well on these large-scale problems.

Other approaches to automated feature selection rely on propositional logic, such as those presented by Mannion (Mannion, 2002) and Batory (Batory, 2005). 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. Filtered Cartesian Flattening 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.

Dudley et al (Dudley et al., 2004) describe a method for implementing automatic self-healing systems. When an integral element of the system architecture becomes disabled, another element is selected by the system that will allow the system to be restored to a functional configuration. Potential elements that could restore the system without detriment to the system are defined by policies. The validity of the selection of an element that matches the constraints defined by these policies are modeled as CSPs. Since Dudley's technique is designed for runtime healing, it must operate quickly and cannot consider resource constraints. The solver searches for any solution, regardless of the resource consumption that would result form its architectural implementation. In contrast, Filtered Cartesian Flattening, can be used to find a solution that is valid and honors resource consumption constraints, which are critical for correctness.

7.2 Approximation Algorithms

Although exact algorithms to solve NP-hard problems have exponential complexity, each NP-hard problem typically has a number of approximation algorithms that can be use to solve it with acceptable optimality. For example, approximation algorithms are commonly used in the shipping industry to solve *Traveling Salesman Problems* (Lin and Kernighan, 1973). Moreover, simple heuristic algorithms, such as *First Fit Decreasing* (Coffman Jr et al., 1996), can be used to effectively solve *Bin-Packing Problems* (Coffman Jr et al., 1996). As we will show in this section, however, there are no existing approximation algorithms that can be applied to optimal feature selection with resource constraints.

Bin-packing approximation algorithms (Coffman Jr et al., 1996; Lodi et al., 1999), such as First Fit Decreasing (Coffman Jr et al., 1996), cannot be applied to architectural feature selection problems because they assume that all of the N items can and must be selected. For example, in Figure 3, there is no way to represent that either `Euclidean` or `IdaSoft` should be put into a bin but not both. That is, bin-packing algorithms can guarantee that resource constraints are not exceeded but they cannot ensure that the architectural constraints in (C) are also simultaneously satisfied.

Knapsack problems (Ibarra and Kim, 1975) also suffer from this same limitation of bin-packing algorithms. Knapsack algorithms (Kellerer and Pferschy, 1999) assume that any combination of the items can be placed into the knapsack. These algorithms can guarantee that the resource limits are not exceeded but again cannot ensure that (C) is simultaneously satisfied. Some knapsack approximation algorithms have been extended with the ability to handle cardinality constraints (Caprara et al., 2000), but cardinality constraints are not sufficient to guarantee (C).

MMKP problems (Akbar et al., 2001), which are the basis of Filtered Cartesian Flattening, are more closely related to architectural feature selection problems. MMKP problems define a group of sets from which at most one item can be selected from each set. For special cases of the architectural feature selection problem where there are no nested variabilities, such as in Figure 5, these algorithms can be applied. When nesting of variabilities occurs, these approximation algorithms (Akbar et al., 2001) are no longer applicable.

For example, the nested choice of general algorithm type (`LDA`, `PCA`, etc.) followed by the choice of the exact variant of the algorithm (`LDA Euclidean` vs. `LDA IdaSoft`) cannot be directly solved with an existing MMKP approximation algorithm. In Section 5, we show how a feature model can be transformed into a format that can be operated on by existing MMKP approximation algorithms to create an approximation algorithm for optimal feature selection problems with resource constraints.

8 Concluding Remarks

To make the sound architectural feature selection decisions, developers need the ability to algorithmically generate architectural variants that optimize desired system properties. A key challenge, however, is that selecting a set of architectural features that maximizes a system capability while adhering to resource constraints is an NP-hard problem. Although there are numerous approximation algorithms for other NP-hard problems, they do not directly support optimal feature selection subject to resource constraints. MMKP approximation algorithms can be applied to specific subsets of optimal feature selection problems, but in general, 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 developers can realistically optimize and evaluate architectural design alternatives.

This paper shows how the *Filtered Cartesian Flattening* approximation technique can be applied to select good feature sets subject to resource constraints. Filtered Cartesian Flattening produces an approximate MMKP representation of a feature selection problem with resource constraints. Each MMKP set contains partial configurations of the architectural feature model and selecting one partial configuration from each MMKP set is guaranteed to produce a correct and complete architectural variant. With this MMKP problem representation, existing MMKP approximation algorithms can be used to solve for an architectural variant that maximizes a specific system property while honoring resource constraints.

Our empirical results show how Filtered Cartesian Flattening can achieve an average of more than 90% optimality for large feature models. Moreover, the results revealed a key problem characteristic, resource tightness ($\frac{AvailableResources}{Avg.ItemResourceConsumption}$), that could be used to predict which problem instances Filtered Cartesian Flattening does not perform well on. These results could be further improved by using more exact MMKP approximation algorithms for the final Filtered Cartesian Flattening phase, such as M-HEU (Akbar et al., 2001).

From our experience developing and evaluating Filtered Cartesian Flattening, we have learned the following lessons:

- The resource tightness metric presented in Section 6.1 is an accurate indicator of whether or not the Filtered Cartesian Flattening technique should be applied to a problem. When the tightness metric value is low, there is little guarantee that Filtered Cartesian Flattening will produce a 90%+ optimal solution.

- 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, Filtered Cartesian Flattening requires between 1-12 seconds for a problems with 1,000-10,000 features. The algorithmic complexity of Filtered Cartesian Flattening is polynomial.
- Once Filtered Cartesian Flattening subdivides a feature model into a number of independent subtrees, these subtrees can be distributed across independent processors to flatten in parallel. The Filtered Cartesian Flattening technique is thus highly amenable to multi-core processors and parallel computing.
- Although other papers (Akbar et al., 2001) have shown that MMKP approximation algorithm solution optimality improves as the number of items and sets grows, Filtered Cartesian Flattening shows a slow degradation in optimality as the number of features grows. When increasing from 1,000 to 10,000 features, we observed a roughly 3% decrease in optimality from 95.54% to 92.56% optimality. In our experiments, K was held constant. We hypothesize that for larger problems, because K was held constant,the drop in optimality is due to the Filtered Cartesian Flattening Cartesian flattening phase being forced to filter out more good partial configurations. K could be increased for larger problems to counteract this decrease in optimality.
- A key attribute of Filtered Cartesian Flattening is its ability to leverage any MMKP solver and any arbitrary filtering strategy for the Cartesian flattening. Simultaneously using and solving problems with multiple filtering strategies and MMKP solvers should provide further guarantees on the optimality of the solution found. The speed of the Filtered Cartesian Flattening algorithm and MMKP solvers makes this multi-pronged solving approach possible.

An implementation of Filtered Cartesian Flattening is included with the open-source GEMS Model Intelligence project and is available from `sf.net/projects/gems`.

APPENDIX

This appendix contains a detailed pseudo-code listing for the Filtered Cartesian Flattening algorithm.

```
cardGroupToXOR:        a function that takes a cardinality group and converts
                       it to an XOR group with K elements using a filtering
                       strategy
cardFeatureToXOR:      a function that takes a Feature with a cardinality
                       expression and converts it to an XOR group with K
                       items
optionalFeatureToXOR:  a function that takes an optional feature and converts
```

```
                      it to an XOR group
NULL_ITEM:            an item with zero resource consumption and value that
                      represents a set that has no items selected

Filtered Cartesian Flattening(RootFeature)                                  (1)
   List Subtrees = new List
   Subtrees.add(RootFeature)
   cut(RootFeature,Subtrees)
   for each Root in Subtrees
      convertToXOR(Root)
   List MMKPSets = new List
   for each Root in Subtrees
      MMKPSets.add(flatten(Root))
   for each MMKPSet in MMKPSets
     if(!MMKPSet.includes(RootFeature))
        MMKPSet.add(NULL_ITEM)
   List solution = MMKPSover.solve(MMKPSets)

cut(Feature, SubtreeRoots)                                                  (2)
  if(Feature.isOptional)
     Feature.parent.children.remove(Feature)
     Feature.parent = null
     SubtreeRoots.add(Feature)
  else
     for each Child in Feature.children
        cut(Child,SubtreeRoots)

convertToXOR(Feature)                                                       (3)
  if(Feature.childrenAreCardGroup)
     cardGroupToXOR(Feature)
  else if(Feature.isOptional)
     optionalFeatureToXOR(Feature)
  else if(Feature.hasCardExpression)
     cardFeatureToXOR(Feature)
  for each Child in Feature.children
    convertToXOR(Child)

List flatten(Feature)                                                       (4)
  List flattened = new List
  if Feature.children.size == 0
     flattened.add(Feature)                                                 (5)
  else if Feature.childrenAreXORGroup
     for each Child in Feature.children
        flattened.addAll( flatten(Child) )                                  (6)
  else if Feature.childrenAreRequired
     flattened.addAll( flattenAll(Feature.children))
  return flattened

List flattenAll(Features)
  List flattened = flatten(Features.first)
  for each Feature in Features
    if(Feature != Features.first)
       flattened = filteredCartesianProduct(flattened,flatten(Feature))     (7)
  return flattened

List filteredCartesianProduct(A,B)
  List product = new List
  for 1 to K                                                                (8)
     choose an item from A
     choose an item from B that satisfies all of A.crossTreeConstraints
     C = new Item(A.partialConfiguration + B.partialConfiguration)
     C.resourceConsumption = A.resourceConsumption + B.resourceConsumption
     C.value = A.value + B.value
     C.crossTreeCons = A.crossTreeCons + B.crossTreeCons
     product.add(C)
  return product
```

References

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

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

Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2007. 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).

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

Caprara, A., Kellerer, H., Pferschy, U., Pisinger, D., 2000. Approximation algorithms for knapsack problems with cardinality constraints. European Journal of Operational Research 123 (2), 333–345.

Coffman Jr, E., Garey, M., Johnson, D., 1996. Approximation algorithms for bin packing: a survey. Approximation algorithms for NP-hard problems table of contents, 46–93.

Coplien, J., Hoffman, D., Weiss, D., Nov.-Dec. 1998. Commonality and variability in software engineering. IEEE Software 15, 37–45.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., 1990. Introduction to Algorithms. MIT.

Dudley, G., Joshi, N., Ogle, D., Subramanian, B., Topol, B., 2004. Autonomic Self-Healing Systems in a Cross-Product IT Environment. Proceedings of the International Conference on Autonomic Computing, 312–313.

Harel, D., June 1987. Statecharts: A visual formalism for complex systems. Science of Computer Programming 8 (3), 231–274.
URL `citeseer.ist.psu.edu/article/harel87statecharts.html`

Ibarra, O., Kim, C., 1975. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. Journal of the ACM (JACM) 22 (4), 463–468.

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

Kellerer, H., Pferschy, U., 1999. A New Fully Polynomial Time Approximation Scheme for the Knapsack Problem. Journal of Combinatorial Optimization 3 (1), 59–71.

Lin, S., Kernighan, B., 1973. An effective heuristic algorithm for the traveling salesman problem. Operations Research 21 (2), 498–516.

Lodi, A., Martello, S., Vigo, D., 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS Journal on Computing 11 (4), 345–357.

Mannion, M., 2002. Using first-order logic for product line model validation.

Proceedings of the Second International Conference on Software Product Lines 2379, 176–187.

Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., Saval, G., 2007. 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. pp. 243–253.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4384187

MOSER, M., JOKANOVIC, D., SHIRATORI, N., 1997. An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences 80 (3), 582–589.

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

Parker, J., Parker, J., 1996. Algorithms for Image Processing and Computer Vision. John Wiley & Sons, Inc. New York, NY, USA.

Phillips, P., Moon, H., Rizvi, S., Rauss, P., 2000. The FERET Evaluation Methodology for Face-Recognition Algorithms.

Pisinger, D., 1995. A minimal algorithm for the multiple-choice knapsack problem. European Journal of Operational Research 83 (2), 394–410.

Sinha, P., Zoltners, A., 1979. The multiple-choice knapsack problem. Operations Research 27 (3), 503–515.

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

White, J., Schmidt, D. C., Benavides, D., Trinidad, P., Ruiz-Cortez, A., September 2008. Automated Diagnosis of Product-line Configuration Errors in Feature Models. pp. 659–668.