To appear in Workshop on Analysis of Software Product Lines (ASPL) 2008, at SPLC 2008

Finding Contradictions in Feature Models

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

Abstract

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

1. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

2. A Run-Time Solution and Perspective

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

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

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

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

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

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

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

3. Feature Models and the BCP Algorithm

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

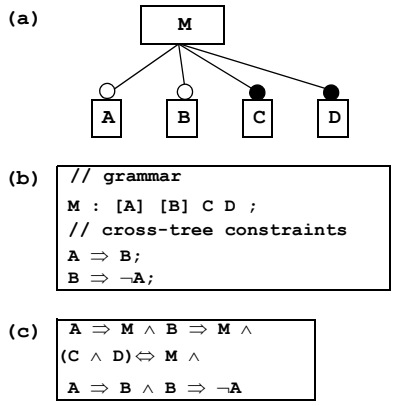

Figure 1 Feature Diagrams

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

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

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

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

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

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

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

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

4. Spin (Simple Promela INtrepreter)

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

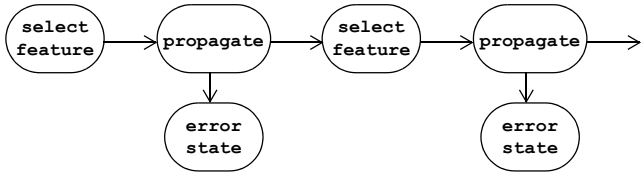

Figure 2 State Machine of Feature Selection and Constraint Propagation

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

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

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

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

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

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

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

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

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

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

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

    do: // loop till you finish selecting all features

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

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

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

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

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

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

Figure 4 Different Feature Models Used In Our Experiments

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

Figure 5 Execution Time of Spin for Different Feature Models

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

5. Incremental Consistency Algorithm

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

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

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

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

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

Figure 6 LookAhead Algorithm

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

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

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

Figure 7 Contradiction Free Algorithm

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

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

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

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

Figure 10 shows how far our optimized ICA algorithm was able to prove contradiction freedom. The BerkeleyDB model has 55 selectable features; in one hour of computation, we were able to prove that it was 5-conflict free. The Violet model has 64 selectable features, and in one hour we were able to prove it was 3-conflict free. In short, we hardly covered any of the state space of these models. We believe the ICA algorithms can prove conflict freedom in models that have about 20 (or fewer) features; ICA seems ineffective in models with substantially larger number of features.

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

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

Figure 8 Incremental Consistency Algorithm

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

Figure 9 Execution Time of ICA for Different Models

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

Figure 10 Coverage of ICA

6. Perspective and Related Work

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

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

6.1. Related Work

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

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

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

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

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

7. Conclusions

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

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

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

8. References

[1] AAAI Workshop on Configuration, 2003. **www2.il-og.com/ijcai-03/**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] **www.spinroot.com**

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

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

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

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

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

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