Software Product Lines

Exercise 10: Product-Line Analyses

ISF (TU Braunschweig) January 2026

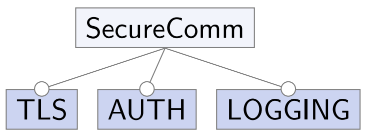
1. **Analysis Strategies for Software Product Lines**

   Define and explain the three primary analysis strategies used in software-product-line engineering. For each strategy, discuss its main characteristics, benefits, and drawbacks using concrete examples. Which strategy would you recommend under which circumstances?

2. **Analyzing Disagreements Between Problem and Solution Space**

   Consider the following product line for secure communication:

Feature Model



TLS → AUTH
LOGGING → TLS

Annotated Source Code

```
class SecureComm {
  //#if TLS
  void sendSecure() {
    // encrypted transmission
  }
  //#endif

  //#if AUTH
  void authenticate() {
    checkUser();
    sendSecure();
  }
  //#endif

  //#if LOGGING
  //#if TLS
  void logSession() {
    Logger.log("TLS session started");
  }
  //#endif
  //#endif
};
```

   (a) What is the problem space of a software product line (in general and in this example)? What is the solution space? What problems can arise when developers only consider one of them when conducting an analysis?

(b) Which configurations are valid in the given problem space?

(c) Which configurations compile successfully in the solution space?

(d) Give one configuration that is valid in the problem space but fails to compile in the solution space.

(e) Give one configuration that is invalid in the problem space but compiles in the solution space.

(f) Propose a sensible fix for the discrepancy between problem and solution space in this example.

3. **Analyzing Feature Mappings in Preprocessor-Based Product Lines**

(a) What is dead code in a preprocessor-based product line? What are superfluous annotations? Explain why it is beneficial to know about the presence of these anomalies in a code base.

(b) Which of both problems occur in the following source code, where, and why? Justify your answer using presence conditions.

Conditionally Compiled `Graph` Code

```java
class Graph {
  //#ifndef DIRECTED
  //#ifdef HYPER
  void edgeVariant1() {
    Set<Node> nodes;
  }
  //#ifdef DIRECTED
  void edgeVariant2() {
    Node a, b;
    a.connectTo(b);
  }
  //#endif
  //#else
  //#ifndef HYPER
  void edgeFallback() {
    Pair<Node, Node> link;
    //#ifdef DIRECTED
    link = new Pair<>(new Node(), new Node());
    //#endif
  }
  //#endif
  //#endif
};
```

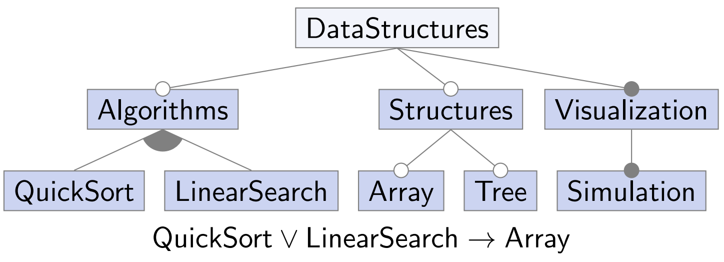
(c) Given any (unpreprocessed) code fragment, how can we automatically detect dead code and superfluous annotations?

(d) Should the problem space be incorporated into the automated detection procedure? If so, how? If not, why not?

4. **Analyzing Variable Code in Preprocessor-Based Product Lines**

Given below is the feature model and code for variable data structures:

Feature Model



Annotated Source Code

```
public class Main {
  //#if Array
  static int[] array = new int[10];
  //#endif

  public static void main(String[] args) {
    //#if Array
    System.out.println(array);
    //#if !Structures
    System.out.println("Structures");
    //#endif
    //#endif
    //#if QuickSort
    System.out.println(sort(array));
    //#endif
  }

  //#if QuickSort
  static int[] sort() {
    int[] newArray = new int[10];
    //#if Simulation
    System.out.println("Statistics");
    //#endif
    return newArray;
  }
  //#endif
}
```

(a) Considering both the problem and solution space, are there dead code fragments or superfluous annotations in this example?

(b) In general, how can unreachable references and conflicting definitions be detected with a SAT solver? Check whether such anomalies appear in this example.

(c) Can you detect any additional errors in the given example that prevent compiling? Which configurations provoke the found errors?

5. **NP-Completeness of SAT: Theory and Practice**

To answer this task, you may read Sections 1, 4, 5, and 10 of the paper attached below:

Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models Is Easy. In SPLC. Carnegie Mellon University, USA, 231–240.

(a) The paper discusses a contradictory phenomenon with respect to the satisfiability problem of propositional logic (SAT) and feature models. Where lies this contradiction, and how do the authors explain it?

(b) According to the authors, how can the 'difficulty' of a SAT instance (i.e., a propositional formula) be characterized? Which SAT instances are particularly difficult? Also refer to the terms *transition phase* and *crossover point*.

(c) Why is it still difficult to determine the *number* of all valid feature configurations by using a SAT solver?

SAT-based Analysis of Feature Models is Easy

Marcilio Mendonca
University of Waterloo, Canada
marcilio@csg.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

Krzysztof Czarnecki
University of Waterloo, Canada
kczarnec@gsd.uwaterloo.ca

Abstract

Feature models are a popular variability modeling notation used in product line engineering. Automated analyses of feature models, such as consistency checking and interactive or offline product selection, often rely on translating models to propositional logic and using satisfiability (SAT) solvers.

Efficiency of individual satisfiability-based analyses has been reported previously. We generalize and quantify these studies with a series of independent experiments. We show that previously reported efficiency is not incidental. Unlike with the general SAT instances, which fall into easy and hard classes, the instances induced by feature modeling are easy throughout the spectrum of realistic models. In particular, the phenomenon of phase transition is not observed for realistic feature models.

Our main practical conclusion is a general encouragement for researchers to continued development of SAT-based methods to further exploit this efficiency in future.

1 Introduction

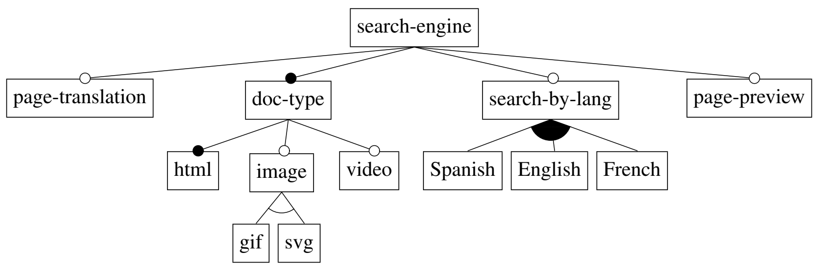
“Variability modeling is a central technique required to put software product line engineering into practice.” [39, p. 88] Indeed, variability modeling and variability models partake in almost all phases and activities of a software product line life cycle: domain analysis, scoping, product selection, product derivation, runtime and evolution. Feature models [29, 17] and their derivatives are a popular notation for variability models, as exemplified by tool offerings of businesses like pure-systems GmbH and BigLever Software Inc., and a plethora of related research in proceedings of the Software Product Lines Conference series.

As a modeling language, feature models undergo a quick maturation process. Tools begin to flourish [12, 30, 4, 45, 31], new applications show up [5], and first voices calling for standardization can be heard [40]. Computer supported variability modeling introduces important benefits over informal modeling. The major advantage is that formal variability models support automation of the built process [17] of family members. Another advantage is possibility of automatic transformation and analysis of models [6].

We look into a broad category of analyses and transformations of feature models: those based on satisfiability solving (SAT) techniques, including consistency checking, finding dead and common features, model entailment and equivalence checking, and interactive and batch configuration. Many research groups [43, 27, 9, 35, 21, 33] have used these techniques in analysis tools observing very good performance even in the case of large-scale models. This fact is somewhat intriguing as the SAT problem, which is the underlying technical problem solved for these analyses, is well known to be intractable [15].

The satisfiability solving community has studied practically interesting subclasses of the SAT problem observing that the hardness is not well distributed across all instances. Many real world problems are either over-constrained (in terms of variability: they have no realizable products) or under-constrained (they have many easily identifiable realizations). These problems are easy to solve for solvers in practice. The difficult problems lie in the middle of the spectrum, in the so called *transition phase* [26].

Our main goal is to make such a hardness study for satisfiability problems that arise in analysis of feature models. We want to identify hard and easy classes and to understand how feature models can be characterized in terms of hardness. Are feature models hard to analyze? Is it well founded to use SAT solvers in implementing analyses for feature modeling tools? Is the efficiency observed by tool builders accidental, or is it a general phenomenon? Can independent experiments confirm efficiency of the analyses cited above?



(*search-by-lang* $\rightarrow$ *page-translation*) $\wedge$ (*page-preview* $\rightarrow \neg$*svg*)

Figure 1. Model of a search engine family

We conclude that large-scale realistic feature models pose no significant difficulty for SAT solvers. Surprisingly even, the *phase transition phenomenon is not observed* in feature modeling. We show that across the entire hardness spectrum, from over-constrained to under-constrained, feature models are easy to analyze for satisfiability solvers.

The potential impact of our work has multiple facets. We contribute towards understanding of what realistic models are (as opposed to what models can be created in the language). Most importantly, we justify the widespread use of SAT solvers and SAT-based systems such as Alloy analyzer in works by others, e.g., [43, 21, 27, 33]. We increase the confidence in using SAT solvers for analyses, and thus we encourage their further use for this purpose.

The primary recipients of this paper are designers of analyses for feature models, builders and vendors of modeling tools, and designers of variability modeling languages. In the long run, we hope to indirectly benefit the end users, i.e., developers applying the aforementioned tools.

We continue introducing feature models in Section 2 and surveying analyses techniques in Section 3. Section 4 formulates our main research question. Section 5 gives background on hardness of the general SAT problem, while Sections 6–7 define the class of realistic feature models and study the hardness of the SAT problem for this class. Section 8 covers threats to validity and Section 9 discusses related work. We conclude in Section 10.

2 Feature Models

Figure 1 depicts a simple feature model of a Web search engine product line. It consists of a *feature tree* (top) and *cross-tree constraints* (bottom-left). In the tree, nodes represent features; edges describe feature relations. A single root node, here search-engine, represents the domain concept being modeled. It is included in every product described by the model. *Optional features* are decorated with a hollow circle, e.g., video. Such features can only be part of products that also contain their parent features. For instance, video can be included in products containing doc-type. In contrast, *mandatory features*, e.g., html, decorated with filled circles, appear in all products containing their parents. Here, html appears in all products containing doc-type.

Feature models allow specification of cardinality relations in the form of $[n, m]$ *feature groups*, which enforce that at least n and at most m features in a group must be present in any product containing their parent ($1 \leq n \leq m$, and m is never greater than the number of feature in the group). Feature groups are illustrated as labeled dashed rectangles in the figure. For instance, features Spanish, English and French form an *or-group* (cardinality [1,3]): at least one of them must be present in any product containing search-by-lang. Features gif and svg form an *exclusive-or-group* (cardinality [1..1]): only one of them can and must be included in products containing image.

Additional constraints often accompany a feature model to complement the relations of the feature tree. We refer to these additional constraints as *cross-tree constraints*. The set of all relations in a feature model is thus specified by conjoining the relations of its feature tree with its cross-tree constraints. There are two cross-tree constraints in Figure 1. The first one, search-by-lang $\rightarrow$ page-translation, reads "search-by-lang requires page-translation". The second one, page-preview $\rightarrow \neg$svg, reads "page-preview excludes svg".

We define *cross-tree constraint ratio* (CTCR) as the ratio of the number of features in the cross-tree constraints to the number of features in the feature tree. The CTCR is usually expressed as a percentage value. For instance, the CTCR of the model in Figure 1 is $30\%(=\frac{4}{13})$.

Individual products conforming to a feature model are specified as sets of its features. For instance $S_1 = \{$search-engine, doc-type, html$\}$ describes a Web search engine system of our product line. Configuration C_1 is *legal*: it does not violate any of the constraints in the model. On the other hand, configuration $C_2 = \{$search-engine, doc-type, html, search-by-lang, page-preview$\}$ is *illegal*. It violates the requirement that presence of search-by-lang requires presence of at least one of Spanish, English or French. Also, the choice of search-by-lang requires inclusion of page-translation according to the cross-tree constraint search-by-lang $\rightarrow$ page-translation, yet page-translation is not part of the specification.

3 Automated Analyses of Feature Models

Existing analyses of feature models fall into two main classes: correctness checking and configuration support. Examples of correctness checking include consistency checking and finding dead or common features. A model is *inconsistent* if its feature constraints prevent any product configurations. A model is incorrect if it contains *dead features* [44], i.e., features that do not belong to any legal product. The dual case are *common features*: features that are shared by all products in the product line. Their presence in feature models is sometimes undesirable. Analyses such as *model entailment* or *equivalence* checking are used to assess the effects of edits on feature models [43, 8]. For example, the application of a refactoring to a model should yield a differently structured, but semantically equivalent model [2].

Automatic analyses are also used to support derivation of products, also known as product configuration. Two kinds of configurations are distinguished: *interactive* and *offline* (batch). In *interactive configuration*, a user makes configuration decisions to derive a product, while the tool guides her in making consistent choices. A configuration system automatically validates and propagates user's decisions in order to enforce that she always eventually reaches a legal configuration. Crucial to interactive configuration is the computation of *valid domains* [25, 23, 27], i.e., the possible values (e.g. true/false or selected/deselected) for features that have not been previously configured by the user. Algorithms for computing valid domains on feature models are known [33, 4, 27]. Slightly different from interactive configuration is *configuration completion* or *batch configuration*. In this case, the configuration system automatically completes a partial configuration without undertaking any further interaction with the user.

The need for effective techniques to analyze and manipulate feature models has attracted the attention of researchers in the field [6]. An important step in this context was the provision of rules for translating feature models to propositional logic [7, 18], which has opened many interesting research opportunities in the use of logic-based systems to reason on feature models, including off-the-shelf technology such as satisfiability solvers.

Table 1 shows the rules for translating a feature model to a propositional formula [7]. The features in the model are the formula variables; the feature tree and the cross-tree constraints represent the formula relations. The root feature is represented by a simple formula (r). There is an implication from every child to its parent. A mandatory feature is implied by its parent. *Inclusive-or* and *exclusive-or* groups are represented by an implication from the parent feature to the respective cardinality relation (fifth and sixth rows; predicate 1-of-n is expanded to a full propositional constraint generalizing xor to n arguments [7]). To obtain the semantics of the entire model, which represents the set of legal configurations, all the formulas for individual syntactic elements are conjoined together with the conjunction of cross-tree constraints .

If the rules in Table 1 are applied to the feature model in Figure 1 the following formula φ is obtained:

$$\text{search-engine} \wedge (\text{page-translation} \rightarrow \text{search-engine}) \wedge \quad (1)$$
$$(\text{doc-type} \rightarrow \text{search-engine}) \wedge \quad (2)$$
$$(\text{search-by-lang} \rightarrow \text{search-engine}) \wedge \quad (3)$$
$$(\text{page-preview} \rightarrow \text{search-engine}) \wedge (\text{html} \rightarrow \text{doc-type}) \wedge \quad (4)$$
$$(\text{image} \rightarrow \text{doc-type}) \wedge (\text{video} \rightarrow \text{doc-type}) \wedge \quad (5)$$
$$(\text{Spanish} \rightarrow \text{search-by-lang}) \wedge \quad (6)$$
$$(\text{English} \rightarrow \text{search-by-lang}) \wedge \quad (7)$$
$$(\text{French} \rightarrow \text{search-by-lang}) \wedge \quad (8)$$
$$(\text{gif} \rightarrow \text{image}) \wedge (\text{svg} \rightarrow \text{image}) \wedge \quad (9)$$
$$(\text{search-engine} \rightarrow \text{doc-type}) \wedge (\text{doc-type} \rightarrow \text{html}) \wedge \quad (10)$$
$$(\text{search-by-lang} \rightarrow (\text{Spanish} \vee \text{English} \vee \text{French})) \wedge \quad (11)$$
$$(\text{image}) \rightarrow (\text{gif} \wedge \neg\text{svg} \vee \neg\text{gif} \wedge \text{svg}) \wedge \quad (12)$$
$$(\text{search-by-lang} \rightarrow \text{page-translation}) \wedge \quad (13)$$
$$(\text{page-preview} \rightarrow \neg\text{svg}) \quad (14)$$

Lines 13–14 represent the cross-tree constraints in the model.

SAT-based Analyses Boolean *satisfiability* is a decision problem to check whether a Boolean formula evaluates to *true* for any of the assignments to its variables. If there is such assignment the formula is said to be *satisfiable*, otherwise it is *unsatisfiable*. For instance, $(a \wedge b)$ is satisfiable as witnessed by the assignment $\{a$=*true*, b=*true*$\}$. Formula $(a \wedge b \wedge \neg a)$ is unsatisfiable: no value for variable a causes the formula to evaluate to *true*.

Table 1. Feature models as Boolean formulas

Feature model relation	**Corresponding formula**
r is the root feature	r
p is parent of feature c	$c \rightarrow p$
m is a mandatory subfeature of p	$p \rightarrow m$
p is the parent of [1..n] grouped features $g_1, \ldots, g_n$	$p \rightarrow (g_1 \vee \ldots \vee g_n)$
p is parent of [1..1] grouped features $g_1, \ldots, g_n$	$p \rightarrow$ 1-of-n$(g_1, \ldots, g_n)$
Cross-tree constraints	already Boolean formulas

SAT solvers are systems used to decide satisfiability. They have been extensively studied for decades. The core decision procedure behind many modern SAT solvers is the DPLL algorithm [19]. State-of-the-art SAT solvers extend it with optimizations techniques allowing them to process large instances efficiently. SAT solvers usually process problems encoded in the *conjunctive normal form* (CNF), i.e., as conjunctions of clauses, where each clause is a disjunction of literals, and a literal is a variable or its negation. Formula $(a \vee b) \wedge (\neg a \vee \neg b)$ is a CNF formula over a and b, comprising two clauses with two literals each.

DPLL is a backtracking search algorithm. It successively assigns Boolean values to variables in the formula, as long as none of the clauses is violated. If an assignment violates one or more clauses, the algorithm backtracks, assigning another value to the offending variable (and perhaps to previously assigned variables if this is insufficient). The procedure stops when all variables have been successfully assigned, or when all assignments have been tried without finding a satisfiable one. Besides search, *unit propagation* is an important part of DPLL. It propagates each variable assignment to other clauses as far as possible, even to clauses not directly related to the propagated assignment.

Since feature models can be translated to propositional logic (and then to CNF), one can use a SAT solver to reason about feature models. For instance, consistency of a model can be decided by checking satisfiability of its corresponding formula. Detecting whether a feature f is dead reduces to checking satisfiability of $\varphi \wedge f$, where φ is the feature model formula. If $\varphi \wedge f$ is unsatisfiable, conclude that feature f is dead; otherwise, it is live. Similar reasoning can be applied to check if f is common: simply check satisfiability of $\varphi \wedge \neg f$, concluding that f is common if the result is negative, or concluding that f not common otherwise. Finally, batch configuration, i.e., configuration completion, requires running a single satisfiability check for the formula φ restricted and simplified with respect to a partial configuration.

Some feature model analyses such as checking model consistency or checking whether a given feature is dead or common require a single SAT check; other analyses may involve several satisfiability checks. For instance, in order to analyze the entire feature model for existence of dead or common features or for computing valid domains [27, 33], several SAT checks are necessary. These SAT checks verify whether selecting or deselecting a given feature violates some constraint or can lead to a legal configuration. Also, techniques exist to perform model entailment or equivalence checking by multiple SAT checks, each checking a formula $\varphi_1 \wedge \varphi_i$, where φ_1 is the formula of the first model and each φ_i is a small conjunction of literals derived from the second model (see [43] for details). Consequently, since making these analyses for the entire model may require thousands of SAT checks, we want each SAT check to be processed in a small fraction of a second.

Researchers [21, 43, 27][1] have observed very good performance of SAT solvers in performing SAT checks on large feature models containing thousands of features. This fact has naturally encouraged researchers to explore other kinds of SAT-based algorithms for feature model analyses. In particular, our own experience with developing such algorithms served to confirm the results of previous research. For instance, we computed consistency checks on models with up to 10,000 features and a large number of cross-tree constraints and in the worst-case scenario the SAT solver took about 0.4 seconds to complete the check. In addition, we developed an algorithm for computing valid domains and observed processing times of about 22 seconds for models with 5,000 features and a fairly large number of cross-tree constraints. Currently, the aforementioned algorithms are used to support analyses and product configuration services at the SPLOT web portal (see http://www.splot-research.org for details).

4 Research Problem

Despite the relative success of the use of SAT solvers to perform automated analyses of feature models, a sensitive issue related to the use of these systems has been consistently neglected. Satisfiability is a difficult computational problem (NP-complete [15]). A SAT solver may take an infeasible amount of time to check satisfiability of certain kinds of instances. It is known that while some classes of instances can be solved efficiently (e.g., 2-SAT—the class of SAT problems consisting of only binary constraints) other classes quickly become intractable with grows of parameters such as number of variables and clauses.

These facts naturally raise the question of whether or not SAT problems derived from realistic feature models can ever become intractable. While previous research has suggested the opposite, researchers have failed to provide a strong evidence of the tractability of those problems. As a consequence, we still do not know precisely whether reported results are accidental or justified.

Therefore, we pose the following research question that shall be addressed in this paper: *Are SAT problems derived from realistic feature models indeed tractable? If so, what evidence supports this fact?*

Addressing this question is crucial to improve the level of confidence in using SAT technology to reason on feature models, especially large ones. In other words, without a good evidence of the efficiency of SAT solvers in the feature

[1]While paper [21] uses small models the authors have confirmed in personal communication that Alloy analyzer (SAT solver) performs well on models with up to 2000 features.

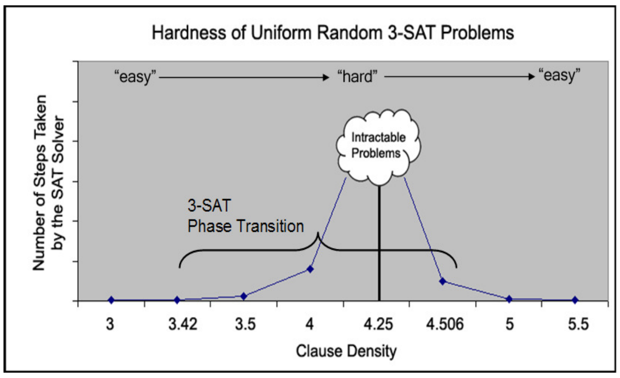


Figure 2. Phase transition of random 3-SAT

modeling domain one cannot fully rely on the use of these systems for a serious purpose.

Methodology. Our approach to answering the research question is as follows. We first analyzed published feature models to determine their structural complexity characteristics, such as the hierarchy branching factor, the CTCR, and the size of the cross-tree constraint clauses. We refer to models with characteristics similar to those of the published models as *realistic*. We then design and run experiments on generated large-scale models that are realistic or even more complex. The experimental design was guided by the prior work studying the hardness of the SAT problem.

5 Hardness of SAT problems

Hardness of SAT problems has been studied in the past primarily with an outlook on improving performance of SAT solvers. In this context, *hardness* is directly related to the number of steps required by a SAT solver to perform satisfiability checks. Obviously, the harder the instance, the higher the number of required steps. Conversely, the easier the instance, the lower the number of required steps. For all known algorithms, the number of required steps is exponential in the worst case, and large worst case instances cannot be solved in feasible time.

As a result, hardness of satisfiability for subclasses of instances has been studied [22, 1, 37, 36, 46]. In particular, researchers have attempted to determine hardness threshold values for k-SAT [36, 20], i.e., the class of SAT problems consisting of CNF clauses containing exactly k literals. These thresholds are related to parameters such as the number of variables and clauses in the problem and represented crossover points in which an "easy" SAT instance becomes "hard", and vice-versa.

An important finding relates hardness of random k-SAT instances to the *phase transition* phenomenon [26, 42, 48]. It characterizes the transition of a random k-SAT instance from a satisfiable to an unsatisfiable state given the variation of a specific order parameter. The main parameter considered in this context is *clause density*, i.e., the ratio α of the number of clauses m to the number of variables n in an instance: $\alpha = \frac{m}{n}$. Probability of a random k-SAT instance to become unsatisfiable grows together with its clause density. The notable finding was that during the phase transition k-SAT instances ($k > 2$) follow an "easy-hard-easy" pattern, i.e., a SAT solver that had been working efficiently starts to struggle as the clause density is increased up to a point that the problem becomes intractable, and as the clause density value continues to grow, the solver starts to perform efficiently again. The peak in hardness coincides with the 50% threshold point (so-called *crossover point*), i.e., the point where random k-SAT instances switch from an "almost always satisfiable" state (underconstrained problem) to an "almost always unsatisfiable" state (overconstrained problem). This transition is known to be sharp and its size varies according to the number of variables in the problem.

For the purpose of our analysis, we are particularly interested in random instances of the 3-SAT problem, as will be argued in the next section. Nowadays, approximate bounds for the phase transition of uniform random 3-SAT problems (simply referred to as *random 3-SAT* from now on) are known. An instance of a random 3-SAT formula is built by selecting three different random variables for each clause and negating each with probability 0.5. The maximum number of clauses is then $8\binom{n}{3}$, where n represents the number of variables in the problem. For random 3-SAT the phase transition occurs for values of α varying from 3.42 to 4.506 as shown in Figure 2. The critical value is ≈ 4.25 that represents the crossover point which, as mentioned, coincides with the peak in hardness of SAT algorithms, i.e., the point where problems are most likely to become intractable. The figure also shows the easy-hard-easy pattern followed by 3-SAT problems as the clause density parameter increases. In practice, these problems can quickly become intractable. For instance, we ran an experiment using a modern SAT solver [11] and a random 3-SAT instance containing 300 variables and 1275 clauses (clause density of 4.25) and observed that the solver did not complete after an hour of processing.

6 SAT Problems Induced by Feature Models

One challenge in analyzing hardness of SAT instances induced by feature models is lack of large scale real models for our analysis. It is known that such models exist [6], but access to them is rarely granted. We were able to compile [32, p. 111] a collection of about 20 small to medium scale models (sizes ranging from 11 to 287 features; available for download at http://fm.gsdlab.org) from papers and published case studies, and we studied their characteristics to support automatic generation of random models for our experiments.

Before we delve into the details of sample generation, let us discuss some relevant properties of feature models. First, it is known that well-formed feature trees are always satisfiable [32, p. 77]. A SAT solver still has to perform the usual search procedure, but this operation takes only linear time in the number of features (variables) [32, p. 107]. This fact is a clear indication of the tractability of SAT problems induced by feature trees; however, conjoining the cross-tree constraints with the feature tree can still make the induced SAT instances hard. Hence, it is important that we consider the structure of the cross-tree constraints. Most importantly, we have observed that in all of the 20 published models we studied, the cross-tree constraints consisted of a mix of binary and ternary CNF clauses. Typically, these clauses resulted from the translation of implication relations involving two or more features (variables) to CNF. For instance, if relation ($a \rightarrow b \wedge c$) is translated to CNF, the resulting formula contains two clauses: $(\neg a \vee b) \wedge (\neg a \vee c)$.

Based on this study of published models, we concluded that realistic models will have a mix of binary and ternary CNF clauses as cross-tree constraints. For the purpose of our experiments, however, we decided to focus on *3-CNF feature models*, or *3-CNF-FMs*: feature models whose cross-tree constraints are limited to be 3-CNF formulas. A SAT problem derived from a 3-CNF-FM is typically harder than those derived from the studied sample of published models since all cross-tree constraints of the 3-CNF-FM are made of ternary clauses. In addition, our experiments use very large models containing up to 10,000 features and with up to 30% CTCR (see Section 2), while most of the published models are usually much smaller and with lower CTCR.

Sample Generation. Our tool generates 3-CNF-FMs in two phases. First, a random feature tree with a specified number of features n is built, with different odds for mandatory, optional, inclusive-OR, and exclusive-OR features. For our models, we indicated an equal likelihood of 0.25 for each type. We also limited a branching factor for the tree by specifying a minimum of 2 and maximum of 10 children per parent feature. This parameter is reasonable for generating realistic trees since branching factors in the published models were usually smaller than 10. The process stops when n features are added to the feature tree.

Second, the tool builds a random 3-CNF formula over a subset of features in the tree, depending on the CTCR. For instance, for a model with 1,000 features and 30% CTCR, 300 distinct variables are selected randomly from the model and combined randomly into ternary CNF clauses. Variables are negated in each clause with 0.5 probability and

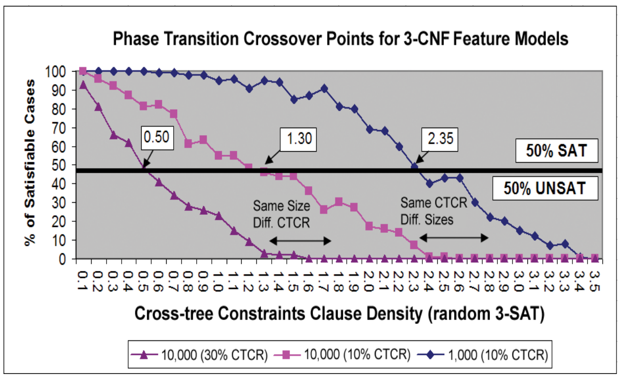


Figure 3. Crossover points for 3-CNF-FMs

identical clauses are not permitted. The number of clauses is controlled by clause density. For instance, given clause density of 2.3 for a model of 1,000-features, we generate 690 ($= 2.3 \times 300$) random ternary clauses. The clause density refers to the density of clauses in the cross-tree constraints not in the formula induced by the entire feature model

Phase Transition of 3-CNF Feature Models. Our first analysis consisted of determining phase transition thresholds for 3-CNF-FMs, i.e., the crossover points in which a 3-CNF-FM instance has 0.5 probability of being (un)satisfiable. Recall from Section 5 that it is near these thresholds that random 3-SAT instances become hard. By determining these thresholds for 3-CNF-FMs, we are indeed finding the clause density values in which those models are most likely to derive hard (and perhaps intractable) SAT problems.

Experiment #1: Crossover points for 3-CNF-FMs. For this experiment, 100 models were generated for each combination of size (1,000, 2,000, 3,000, 5,000 and 10,000), CTCR (10%, 20% and 30%) and clause density (from 0.1 to 3.5 in increments of 0.1). We want to examine how the hardness of feature model SAT instances increase as new clauses are added to the cross-tree constraints. In total, 52,500 models were generated during the experiment (some of which are available online (http://fm.gsdlab.org/). The SAT solver (SAT4J [11]) was given 30 seconds to complete satisfiability checks on generated models.

Due to space concerns, we report only the most essential findings here. For a complete list of the crossover points identified for 3-CNF-FMs, please refer to [32, p. 141].

Figure 3 shows the results of the crossover-point computation for three classes of feature models containing 1,000 (10% CTCR) and 10,000 (10% and 30% CTCR) features. The graph shows the percentage of satisfiable models (vertical axle) as the clause density of the 3-SAT cross-tree constraint formula increases from 0.1 to 3.5 (horizontal axle). Each point in the graph has been computed using 100 generated models.

The crossover-point line is shown as a horizontal line in the 50% satisfiability point on the vertical axis. The three crossover points indicated in the figure (0.50, 1.30 and 2.35) represent the median density values within the range in which at most 70% and at least 30% of the models analyzed represented satisfiable instances. Each of these points refers to a specific class of models with fixed size and CTCR. For instance, consider the models with 1,000 features and 10% CTCR (rightmost series). At density values 2.0 and 2.7 no more than 70% and no less than 30%, respectively, of the models analyzed were satisfiable. The median value was then the average of the two middle values 2.3 and 2.4 which resulted in the crossover point of 2.35 (see this value in Figure 3). This value suggests that if the CNF formula corresponding to a feature tree containing 1,000 nodes is conjoined to a random 3-SAT formula containing 100 of the feature tree variables (10% CTCR) and 235 clauses (density of 2.35), the resulting SAT instance has 50% of chance of being satisfiable.

Results. The first finding in this experiment is that *for a fixed model size, the higher the CTCR the earlier the feature model instance reaches the crossover point and becomes unsatisfiable*. Consider the two models in Figure 3 containing 10,000 features. The one with higher CTCR (30%) reaches the crossover point at density 0.5 while the other with CTCR of 10% reaches the same point at a higher density of 1.30. This result was expected since increasing the number of variables and clauses in the cross-tree constraints also increases clause density of the formula induced by the model.

Second, *for a fixed CTCR, the larger the model the earlier the feature model instance reaches the crossover point and becomes unsatisfiable*. This finding is interesting since the number of variables in the cross-tree constraints is equally proportional to the size of the models (i.e., fixed CTCR), but yet larger models are more sensitive to the addition of clauses to the cross-tree constraints than smaller ones. Recall that for 3-SAT problems the crossover point happens about density 4.25 no matter the size of the problem (number of variables). Yet, the size of the feature tree influences the crossover thresholds of 3-CNF-FMs with the same CTCR value. Figure 3 illustrates this scenario. Given two models with CTCR of 10% and containing 1,000 and 10,000 features, the one with higher number of features reaches the crossover point first at 1.30 against density 2.35 of the other (see these values in the figure).

The most important result of this experiment is determining hardness thresholds for 3-CNF-FMs, i.e., critical density values for which models should induce hard SAT problems. These thresholds will be used in our hardness tests shown in the next section.

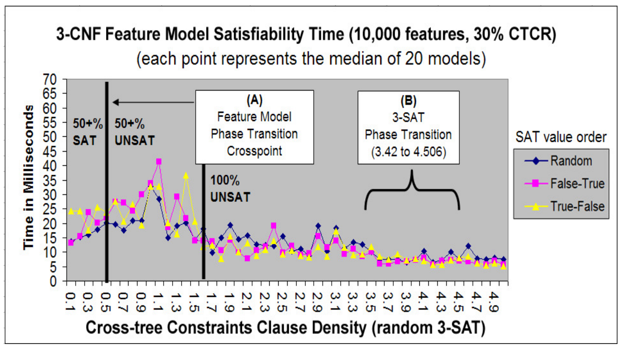


Figure 4. Times of satisfiability checks

7 Hardness of 3-CNF Feature Models

In the second experiment, we examine the hardness of SAT instances derived from 3-CNF-FMs taking into account the crossover points identified for those in the previous experiment. We focused the analysis on very large feature models with 10,000 features and the highest CTCR in the previous experiment (30% CTCR). We believe that these two parameters are large enough to generate SAT instances that are much harder than the vast majority of realistic feature models.

In addition, since typical SAT-based feature model analyses such as detecting dead and common features and computing valid domains usually require the solver to use a particular value order during the search procedure, we decided to also test different value orders for the SAT solver. For instance, it makes sense to set up the solver with value order $\{true, false\}$ when computing dead features since the goal in this analysis is to find as many *true* variables as possible in each solution provided by the solver. By doing so, one eliminates all these variables from the next rounds of satisfiability checks since it is known they are not dead. The same rationale applies to common features, but with the opposite value order, i.e., $\{false, true\}$. In other situations, the value order might not be important; thus, we also tested random value orders.

Experiment #2: Tractability of 3-CNF-FMs. For this experiment, we used 20 feature models with 10,000 features and 30% CTCR for each density value within the range from 0.1 to 5.0 in increments of 0.1. In total, 1,000 models were generated during the experiment. The SAT solver (SAT4J [11]) was given 30 seconds to complete satisfiability checks. It was configured with three different value orders: $\{true, false\}$, $\{false, true\}$, and random. The variable order indicated to the SAT solver was obtained by a depth-first traversal on the feature tree.

Results. The crossover point for models with 10,000 features and 30% CTCR was 0.5, as shown in Figure 3. Figure 4 shows the median running time of SAT4J performing satisfiability checks for various clause density values and for three different value orders. As it can be observed in the figure, the SAT solver was extremely efficient regardless of the density or the value order considered. That is, the median running time for satisfiability checks was never higher than 42 ms. The worst absolute running times were *never higher* than 400 ms in any of the test cases. In particular, there was no decline in performance of the SAT solver for instances near the critical density value of 0.5, the feature model crossover point (see label (A) in the figure). This finding provides a strong evidence that *SAT problems derived from 3-CNF-FMs can be solved efficiently by a standard SAT solver*.

Another interesting point illustrated in Figure 4 (see label (B)) shows that when the cross-tree constraint formula (in isolation) reaches its phase transition (from 3.42 to 4.506) and becomes computationally hard to solve, the SAT solver observes no decline in performance. Therefore, we can conclude that *when an intractable 3-SAT formula is conjoined to a feature tree the resulting feature model formula is unsatisfiable and easily solvable*.

8 Threats to Validity

Let us now briefly discuss threats to validity of our work.

Validity of the experimental approach. The experimental analysis of performance, by its very nature, cannot prove the timing behavior of techniques for all possible cases. An alternative would be to study complexity-theoretic properties of feature models as constraint systems. Due to inherent difficulty of satisfiability, an experimental approach has been previously adopted by the SAT solving community [22, 37, 46]. We follow that adoption here.

Applicability of results for realistic models. Generating models that resemble published models can be risky for our analysis of problem hardness since we would assess the expected case behavior. It is more relevant to build a class of models that share essential properties of real models (e.g., consisting of a feature tree and cross-tree constraints), but that can induce much harder SAT problems than most of the realistic models. This class would inform us about *worst-case* performance for human-constructed models. We have mitigated this threat by generating instances that are harder than those seen in published models, and shown that even these instances are easily solvable.

Limitations of our conclusion. We have concluded that satisfiability problems induced by feature models are easy to solve; however, since feature models are complete with respect to propositional logics [13] one can certainly encode intractable SAT problems in feature models. In fact, this can be done easily if the children-per-parent parameter is increased to a point where the feature tree has depth 1.

It is important to emphasize that our conclusions only apply to "realistic models," as defined in Section 6 and reflected in our generation procedure, and not to all models that can be designed in the language. In particular, our class of realistic models was inspired by a sample of published feature models, and it is unclear how well this sample reflects the models that are used in actual industrial practice.

Also note that if hard time bounds are needed, then it is more suitable to consider alternative analysis techniques such as BDDs, which may be slower on average, but guarantee a fast response time, if the BDD for the problem has been constructed up front (offline). This is possible for some use cases, for example for interactive configuration [25, 24].

Applicability to other analyses Our results apply to SAT-based analyses that work with relatively small adjustments of the CNF formula induced by the feature model, such as adding few new clauses. Consistency checking, detection of dead or common features, and computation of valid domains are analysis that require no or small adjustments. If a method applies major manipulations to the formula or if it uses a different decision/optimization problem (such as MAX-SAT [38, chpt. 9]) then the results do not carry over automatically.

9 Related Work

We have extensively reported about SAT-based techniques for analysis of feature models throughout the paper. For the sake of completeness, we mention here relevant sources that do not use SAT directly.

Automated analyses were also applied to *extended* feature models, i.e., models that contain non-Boolean attributes associated to features [10]. Non-Boolean models are naturally described as a constraint satisfaction problem. Similar analysis are possible on those models such as checking consistency or computing valid domains. Constraint *optimization* systems have been used to detect and repair errors in inconsistent configurations [47]. It would be interesting to carry out a similar kind of tractability analysis to ours for systems based on constraint solving.

Alternative approaches are based on Binary Decision Diagrams (BDDs) [14, 3]. BDDs are efficient for some algorithms often difficult for SAT solvers: counting legal configurations, computing valid domains [24], or reverse engineering models [18]. Observe, that one can hardly use a SAT solver to count legal configurations efficiently. Similarly, no SAT-based techniques are known for reverse engineering. A typical advantage of BDD based methods is *guaranteed* efficient response time of analyses used in user interaction. Recall that the worst case response time for SAT solvers is exponential.

BDDs suffer from intractability, too. There the intractability, manifested in space rather than time, is shifted to the compilation phase performed offline. In order to address it, variable ordering heuristics for compiling feature models have been studied [34, 32]. Currently, it is known that models with up to 2,000 features (for CTCR of 10%) can be successfully compiled to BDDs.

Analyses tools that compare different reasoning techniques for feature models are available [9][32, p. 145]. In such tools, one can easily incorporate new techniques and reasoning algorithms and compare their strengths and weaknesses.

10 Conclusion and Future Work

We have presented a brief survey of analysis methods for feature models and an extensive experimental study of efficiency of SAT-based methods for analyses. We have observed that these analyses essentially reduce to satisfiability problems; thus, it suffices to study this more abstract problem to obtain conclusions about the analyses.

Our main technical conclusion is that SAT instances induced by feature models are easy for solvers. This finding has been demonstrated by (1) a series of experiments analyzing parameter combinations that identify crossover points, and (2) by studying hardness of randomly generated instances in a large interval around these points. Our instances were harder than realistically met models. Surprisingly, we have observed no phase transition phenomenon for problems induced by realistic feature models.

Our findings confirm and justify the high-performance of SAT solvers in analyses observed by independent research groups [43, 21, 28]. Indeed SAT solving scales well for these problems, even for models containing up to 10,000 features. There is now experimental evidence indicating that it will continue to scale well for analyses yet to be developed. We believe that our findings will encourage the development of new SAT-based algorithms in the future.

One way to further develop the work presented here is to analyze the problem theoretically. Perhaps the class of realistic feature models can be reduced to one of the PTIME-decidable subclasses of satisfiability described by Shaefer [41][16, p. 52].

References

[1] D. Achlioptas, L. Kirousis, E. Kranakis, and D. Krizanc. Rigorous results for random (2 + p)-SAT. *Theoretical Computer Science*, 265(1):109–129, 2001.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06*, pages 201–210. ACM, 2006.

[3] H. R. Andersen. *Binary Decision Diagrams*. Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, 1997. Lecture notes for 49285 Advanced Algorithms E97, `http://www.itu.dk/people/hra/notes-index.html`.

[4] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004. Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.

[5] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *LNCS*, pages 692–706. Springer, 2006.

[6] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[7] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines, 9th Int. Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[8] D. Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models. A Framework for Developing Automated Tool Support.* PhD thesis, University of Seville, Spain, June 2007. Available from `http://www.lsi.us.es/~dbc/dbc_archivos/pubs/benavides07-phd.pdf`.

[9] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes. FAMA: Tooling a framework for the automated analysis of feature models. *VAMOS 2007*, pages 129–134, 2007.

[10] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005*, LNCS. Springer, 2005.

[11] D. L. Berre, A. Parrain, O. Roussel, and L. Sais. *SAT4J: A satisfiability library for Java*, 2005. `http://www.objectweb.org/phorum/download.php/16,291/sat4j-D-Le-Berre.pdf`.

[12] D. Beuche. pure::variants Eclipse Plugin. User Guide. pure-systems GmbH. Available from `http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf`, 2004.

[13] Y. Bontemps, P. Heymans, P. Schobbens, and J. Trigaux. Generic semantics of feature diagrams variants. In *Features Interactions in Telecommunications and Software Systems VIII(ICFI'05)*, pages 58–77, Leicester, UK, Jun 2005.

[14] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[15] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[16] N. Creignou, S. Khanna, and M. Sudan. *Complexity of Classifications of Boolean Constraint Satisfaction Problems*. SIAM Monographs on discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), 2001.

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

[18] K. Czarnecki and A. Wasowski. Feature models and logics: There and back again. In *Proceedings of 10th Int. Software Product Line Conference (SPLC 2007)*. IEEE Press, 2007.

[19] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Commun. ACM*, 5(7):394–397, 1962.

[20] E. Friedgut. Sharp thresholds of graph properties, and the k-SAT problem. *J. Amer. Math. Soc*, 12:1017–1054, 1999.

[21] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in Alloy. In *First Alloy Workshop*, pages 71–80, Portland, United States, November 2006.

[22] A. Goerdt. A threshold for unsatisfiability. *J. Comput. Syst. Sci.*, 53(3):469–486, 1996.

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

[24] T. Hadzic, R. M. Jensen, and H. R. Andersen. Calculating valid domains for BDD-based interactive configuration. *CoRR*, abs/0704.1394, 2007.

[25] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO Conference*, pages 131–138. DTU-tryk, June 2004.

[26] B. A. Huberman and T. Hogg. Phase transitions in artificial intelligence systems. *Artif. Intell.*, 33(2):155–171, 1987.

[27] M. Janota. Do SAT solvers make good configurators? In *Proceedings of the First Workshop on Analyses of Software Product Lines*, September 2008.

[28] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In *Software Product Line Conference, 2007. SPLC 2007. 11th Int.*, pages 13–22, 2007.

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

[30] C. W. Krueger. Software mass customization. White paper, Oct. 2001.

[31] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 55–59, New York, NY, USA, 2005. ACM.

[32] M. Mendonca. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, School of Computer Science, University of Waterloo, Jan 2009. Available from `http://hdl.handle.net/10012/4201`.

[33] M. Mendonca. SPLOT (Software Product Line Online Tools) website, 2009. Available at: `http://www.splot-research.org`.

[34] M. Mendonca, A. Wasowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *Int. Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 13–22, 2008.

[35] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *Requirements Engineering, IEEE Int. Conference on*, 0:243–253, 2007.

[36] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on AI - American Association for Artificial Intelligence*, pages 459–465, 1992.

[37] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. 2+p-SAT: Relation of typical-case complexity to the nature of the phase transition, 1999.

[38] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[39] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[40] S. Segura and A. Ruiz-Cortéz. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *Third Int. Workshop on Variability Modeling of Software-intensive System (VAMOS)*, 2009.

[41] T. J. Shaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226. ACM Press, May 1978.

[42] B. M. Smith. Locating the phase transition in constraint satisfaction problems. In *Artificial Intelligence*, 1994.

[43] T. Thum, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *31th Int. Conference on Software Engineering (ICSE)*. IEEE, May 2009.

[44] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. Isolated features detection in feature models. In *CAiSE Short Paper Proceedings*, 2006.

[45] P. Trinidad, D. Benavides, A. Ruiz-Cortes, S. Segura, and A. Jimenez. FAMA framework. *Software Product Line Conference, 2008. SPLC '08. 12th Int.*, pages 359–359, Sept. 2008.

[46] B. Vandegriend and J. Culberson. The $G_{n,m}$ phase transition is not hard for the Hamiltonian cycle problem. *Journal of Arti Intelligence Research*, 9:9–219, 1998.

[47] J. White, D. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortes ands. Automated diagnosis of product-line configuration errors in feature models. *Software Product Line Conference, 2008. SPLC '08. 12th Int.*, pages 225–234, Sept. 2008.

[48] K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *CoRR*, cs.AI/0004005, 2000.