Automated Diagnosis of Product-line Configuration Errors in Feature Models *

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

D. Benavides and P. Trinidad and A. Ruiz–Cortés
Dept. of Computer Languages and Systems
University of Seville
Avda. de la Reina Mercedes s/n
B-41012 Seville, Spain
Email:{benavides, ptrinidad, aruiz}@us.es

Abstract

Feature models are widely used to model software product-line (SPL) variability. SPL variants are configured by selecting feature sets that satisfy feature model constraints. Configuration of large feature models can involve multiple stages and participants, which makes it hard to avoid conflicts and errors. New techniques are therefore needed to debug invalid configurations and derive the minimal set of changes to fix flawed configurations.

This paper provides three contributions to debugging feature model configurations: (1) we present a technique for transforming a flawed feature model configuration into a Constraint Satisfaction Problem (CSP) and show how a constraint solver can derive the minimal set of feature selection changes to fix an invalid configuration, (2) we show how this diagnosis CSP can automatically resolve conflicts between configuration participant decisions, and (3) we present experiment results that evaluate our technique. These results show that our technique scales to models with over 5,000 features, which is well beyond the size used to validate other automated techniques.

1 Introduction

Software Product-Lines (SPLs) are a technique for creating software applications composed from reusable parts that can be re-targeted for different requirement sets. For example, in the automotive domain, an SPL can be created that allows a car's software to provide Anti-lock Braking System (ABS) capabilities or simply standard braking. Each unique configuration of an SPL is called a *variant*.

SPL variants cannot be constructed arbitrarily, *e.g.*, a car cannot have both ABS and standard braking software controllers. A key step in building an SPL is therefore creating a model of the SPL's variability and the constraints on variant configuration. An effective technique for capturing these configuration constraints is *feature modeling* [14], which documents SPL variability and configuration rules via *features*. Each feature represents an increment in product functionality. A feature model can capture different types of variability, ranging from *SPL variability* (*e.g.*, variations in customer requirements) to *software variability* (*e.g.*, variations in software implementation)[16].

SPL variants can be specified as a selection or configuration of features. Feature models of SPLs are arranged in a tree-like structure where each successively deeper level in the tree corresponds to a more fine-grained configuration option for the product-line variant, as shown by the feature model in Figure 1. The parent-child and cross-tree relationships capture the constraints that must be adhered to when selecting a group of features for a variant.

Existing research has focused on ensuring that features chosen from feature models are correct and consistent with the SPL and variant requirements. For example, work has been done on using boolean circuit satisfiability techniques [15] or Constraint Satisfaction Problems (CSPs) [7, 21] to automate the derivation of a feature set that meets a requirement set. Numerous tools have also been developed, such as Big Lever Software Gears [9], Pure::variants [8], FeAture Model Analyser (FAMA) [6], and the Feature Model Plug-in [10], to support the construction of feature models and correct selection of feature configurations.

Regardless of what tools and processes are used to configure SPL variants, however, there is always the possibility that mistakes will occur. For example, large SPLs often use *staged configuration* [11, 12], where features are selected in multiple stages to form a complete configuration iteratively, rather than choosing all features at once. At a late stage in the configuration process, developers may realize that a critically needed feature cannot be selected due to one or numerous decisions in some previous stages. It is hard to debug a configuration to figure out how to change decisions in previous stages to make the critical feature selectable [5].

Another challenging situation can arise when multiple participants are involved in the feature selection process and their desired feature selections conflict. For example, hardware developers for an automobile may desire a lower cost set of Electronic Control Units (ECUs) that cannot support

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

the features needed by the software developer's embedded controller code. In these situations, methods are needed to evaluate and debug conflicts between participants. Methods are also needed to recommend modifications to the participants feature selections to make them compatible.

Although prior research has shown how to identify flawed configurations [4, 15], conventional debugging mechanisms cannot pinpoint configuration errors and identifying corrective actions. More specifically, techniques are lacking that can take an arbitrary flawed configuration and produce the minimal set of feature selections and deselections to bring the configuration to a valid state. This paper focuses on addressing these gaps in existing research.

Solution overview and contributions. Our approach to debugging feature model configurations transforms an invalid feature model configuration into a Constraint Satisfaction Problem (CSP) [20] and then uses a constraint solver to derive the minimal set of feature selection modifications that will bring the configuration to a valid state. We call this constraint-based diagnostic approach, *Configuration Understanding and REmedy (CURE)*. This paper shows how CURE provides the following contributions to work on debugging errors in feature model configurations:

1. We provide a CSP-based diagnostic technique, inspired by [19] that can pinpoint conflicts and constraint violations in feature models
2. We show how CURE can remedy a configuration error by automatically deriving the minimal set of features to select and deselect
3. We provide mechanisms for using CURE to cost-optimally mediate conflicting configuration participant feature selection desires
4. We show how CURE allows stakeholders to debug a configuration error or conflict from different viewpoints
5. We provide empirical results showing that CURE is scalable enough to support industrial SPL feature models containing over 5,000 features.

The remainder of the paper is organized as follows: Section 2 describes the challenges of diagnosing configuration errors and conflicts in SPLs; Section 3 presents the CURE CSP-based technique for diagnosing configuration errors and conflicts; Section 4 shows how CURE can be extended to support conflict mediation, multi-viewpoint debugging, and faster diagnosis times; Section 5 presents empirical results demonstrating the ability of CURE to scale to feature models with thousands of features; Section 6 compares CURE with related research; and Section 7 presents concluding remarks.

2 Challenges of Debugging Feature Model Configurations

This section evaluates different challenges that arise in realistic configuration scenarios; Section 3 describes our solutions to these challenges.

2.1 Challenge 1: Staged Configuration Errors

Staged configuration is a configuration process whereby developers iteratively select features to reduce the variability in a feature model until a variant is constructed. Czarnecki et al. [11, 12] use the context of software supply chains for embedded software in automobiles to demonstrate the need for staged configuration. In the first stage, software vendors provide software components that can be provided in different configurations to actuate brakes, control infotainment systems, etc. In the second stage, hardware vendors of the Electronic Control Units (ECUs) that the software runs on must provide ECUs with the correct features and configuration to support the software components selected in the first stage.

The challenge with staged configuration is that feature selection decisions made at some point in time T have ramifications on the decisions made at all points in time $T' > T$. For example, it is possible for software vendors to choose a set of software component features for which there are no valid ECU configurations in the second configuration stage. Identifying the fewest number of configuration modifications to remedy the error is hard because there can be significant distance between T and T'.

This challenge also appears in larger models, such as those for software to control the automation of continuous casting in steel manufacture [17]. In large-scale models, configuration mimics staged configuration since developers cannot immediately understand the ramifications of their current decisions. At some later decision point, critical features that developers need may no longer be selectable due to some previous choice. Again, it is hard to identify the minimal set of configuration decisions to reverse in this scenario. Section 3 describes how CURE addresses this challenge.

2.2 Challenge 2: Mediating Conflicts

In many situations the desired features and needs of multiple stakeholders involved in configuring an SPL variant may conflict. For example, when configuring automotive systems, software developers may want a series of software component configurations that cannot be supported by the ECU configurations proposed by the hardware developers. To each party, their individual needs are critical and finding the middle ground to integrate the two is hard.

Another conflict scenario arises when configuration decisions made for an SPL variant must be reconciled with constraints of the legacy environment in which it will run. For example, when configuring automotive software for next year's car model, a variant may initially be configured to provide the most desired customer features, such as digital infotainment. New model cars are rarely complete redesigns, however, so developers must determine out how to

run new software configurations on existing ECU configurations from previous models. If the new software configuration is not compatible with the legacy ECU configuration, developers must derive the lowest cost set of modifications to either the new software or the legacy ECU configuration. Section 4.3 describes how CURE addresses this challenge by diagnosing the superset of the desired conflicts and leveraging an alternate CSP optimization goal.

2.3 Challenge 3: Viewpoint-dependent Errors

The feature labeled as the source of an error in a feature model configuration may vary depending on the viewpoint used to debug it. In the feature model shown in Figure 1, for example, if a configuration is created that includes both *Non-ABS Controller* and *1 Mbit/s CAN Bus*, either feature can be viewed as the feature that is the source of the error. If we debug the configuration from the viewpoint that

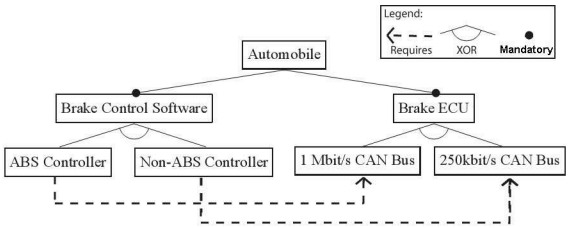

Figure 1: Simple Feature Model for an Automobile

software trumps ECU hardware decisions, then the *1 Mbit/s CAN Bus* feature is the error. If we assume that ECU decisions precede software, however, then the *Non-ABS Controller* feature is the error.

A feature model may therefore require debugging from multiple viewpoints since diagnosing the feature that causes an error in a feature model depends on the viewpoint used to debug it. For small feature models, debugging from different viewpoints is relatively simple. When feature models contain hundreds or thousands of features, the complexity of diagnosing a configuration from multiple viewpoints increases greatly. Section 4.2 describes how CURE addresses this challenge by specifying feature selections that cannot be modified by the solver during diagnosis.

3 Configuration Error Diagnosis

Our solution approach, called Configuration Understanding and REmedy (CURE), is based on creating automated SPL variant diagnosis tools. Developers can use these tools to identify the minimal set of features to select or deselect to transform an invalid configuration into a valid configuration. Moreover, depending on the input provided to CURE, a flawed configuration can be debugged from different viewpoints or conflicts between multiple stakeholder decisions in a configuration process can be mediated.

The key component of CURE is the application of a CSP-based error diagnostic technique. In prior work, Benavides et al. [7] have shown how feature models can be transformed into CSPs to automate feature selection with a constraint solver [13]. Trinidad et al. [19] subsequently described how to extend this CSP technique to identify *full mandatory features*, *void features*, and *dead feature models* using Reiter's theory of diagnosis [18]. This section presents an alternate diagnostic model for deriving the minimum set of features that should be selected or deselected to eliminate a conflict in a feature configuration.

3.1 Background: Feature Models and Configurations as CSPs

A CSP is a set of variables and a set of constraints over those variables. For example, $A+B \leq 3$ is a CSP involving the integer variables A and B. The goal of a constraint solver is to find a valid *labeling* (set of variable values) that simultaneously satisfies all constraints in the CSP. ($A = 1$, $B = 2$) is thus a valid labeling of the CSP.

To build the CSP for the error diagnosis technique, we construct a set of variables, F, representing the features in the feature model. Each configuration of the feature model is a set of values for these variables, where a value of 1 indicates the feature is present in the configuration and a value of 0 indicates it is not present. More formally, a configuration is a labeling of F, such that for each variable $f_i \subset F$, $f_i = 1$ indicates that the i_{th} feature in the feature model is selected in the configuration. Correspondingly, $f_i = 0$ implies that the feature is not selected.

Given an arbitrary configuration of a feature model as a labeling of the F variables, developers need the ability to ensure the correctness of the configuration. To achieve this constraint checking ability, each variable f_i is associated with one or more constraints corresponding to the configuration rules in the feature model. For example, if f_j is a required subfeature of f_i, then the CSP would contain the constraint: $f_i = 1 \Leftrightarrow f_j = 1$.

Configuration rules from the feature model are captured in the constraint set C. For any given feature model configuration described by a labeling of F, the correctness of the configuration can be determined by seeing if the labeling satisfies all constraints in C. A more detailed description of the steps for transforming a feature model to a CSP are described in [7].

3.2 Configuration Diagnostic CSP

When diagnosing configuration conflicts, developers need a list of features that should be selected or deselected to make an invalid configuration a valid configuration. The output of CURE is this list of features to select and deselect, as shown in Figure 2.

In Step 1 of Figure 2, the rules of the feature model and the current invalid configuration are transformed into a CSP. For example, $o_1 = 1$ because the *Automobile* feature

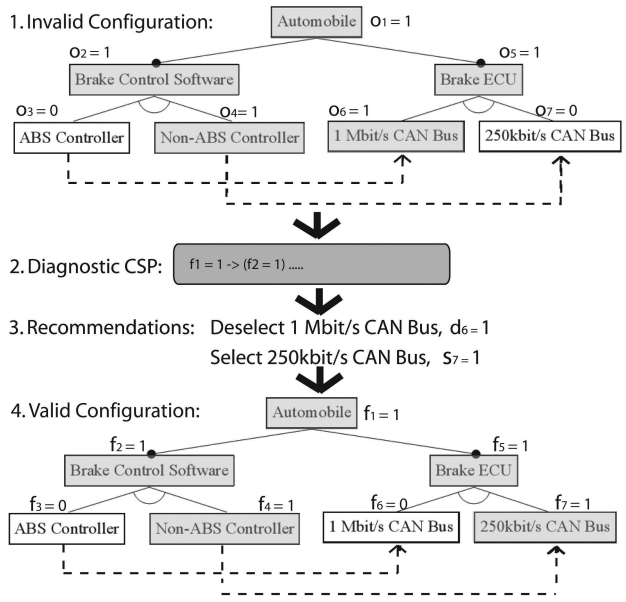

Figure 2: Diagnostic Technique Architecture for CURE

is selected in the current invalid configuration. In Step 2, the solver derives a labeling of the diagnostic CSP. Step 3 takes the output of the CSP labeling and transforms it into a series of recommendations of features to select or deselect to turn the invalid configuration into a valid configuration. Finally, in Step 4, the recommendations are applied to the invalid configuration to create a valid configuration where each variable f_i equals 1 if the corresponding feature is selected in the new and valid configuration. For example, $f_7 = 1$, meaning that the *250 Kbit/s CAN Bus* is selected in the new valid configuration.

To enable the constraint solver to recommend features to select and deselect, two new sets of recommendation variables, S and D, are introduced to capture the features that need to be selected and deselected, respectively, to reach a valid configuration. For example, a value of 1 for variable $s_i \subset S$ indicates that the feature f_i should be added to the current configuration. Similarly, $d_i = 1$ implies that the feature f_i should be removed from the configuration.

Thus, for each feature $f_i \subset F$, there are variables $s_i \subset S$ and $d_i \subset D$. After the diagnosis CSP is labeled, the values of S and D serve as the output recommendations to the user as to what features to add or remove from the current configuration, as shown in Table 1. This table shows the complete inputs and outputs to diagnose the invalid configuration scenario shown in Figure 2.

The next step is to allow developers to input their current configuration into the solver for diagnosis. Rather than directly setting values for the variables in F, developers use a special set of input variables called the *observations*, which are contained in the set of variables O. For each feature

Variables	
Variable Explanations	$f_i \subset F$: feature variables for the valid configuration that will be transitioned to; $o_i \subset O$: the features selected ($o_i = 1$) in the current invalid configuration; $s_i \subset S$: features to select ($s_i = 1$) to reach the valid configuration; $d_i \subset D$: features to deselect ($d_i = 1$) to reach the valid configuration
Inputs	
Current Config.	$o_1 = 1, o_2 = 1, o_3 = 0, o_4 = 1, o_5 = 1, o_6 = 1, o_7 = 0$
Feature Model Rules	$f_1 = 1 \Leftrightarrow (f_2 = 1), f_1 = 1 \Leftrightarrow (f_5 = 1)$, $f_2 = 1 \Rightarrow (f_3 = 1) \oplus (f_4 = 1)$, $f_5 = 1 \Rightarrow (f_6 = 1) \oplus (f_7 = 1)$, $(f_6 = 1) \vee (f_7 = 1) \Rightarrow (f_5 = 1)$, $(f_3 = 1) \vee (f_4 = 1) \Rightarrow (f_2 = 1)$, $f_3 = 1 \Rightarrow (f_6 = 1), f_4 = 1 \Rightarrow (f_7 = 1)$
Diagnostic Rules	$(f_i \subset F \mid \{(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0), (f_i = 0) \Rightarrow (o_i = 0 \oplus d_i = 1) \wedge (s_i = 0)\})$
Outputs	
Features to Select	$s_1 = 0, s_2 = 0, s_3 = 0, s_4 = 0, s_5 = 0, s_6 = 0,$ **$s_7 = 1$**
Features to Deselect	$d_1 = 0, d_2 = 0, d_3 = 0, d_4 = 0, d_5 = 0,$ **$d_6 = 1$**, $d_7 = 0$
New Valid Config.	$f_1 = 1, f_2 = 1, f_3 = 0, f_4 = 1, f_5 = 1, f_6 = 0, f_7 = 1$

Table 1: Diagnostic CSP Construction

f_i present in the current flawed configuration, $o_i = 1$; if f_i is not selected in the current invalid configuration, $o_i = 0$. Table 1 shows how observations capture the current invalid configuration provided as input to the solver. Observations can also be made for a correct configuration, in which case CURE will state that no changes are needed. The rest of this paper assumes that the observations represent an invalid configuration.

To diagnose the CSP, we want to find an alternate but valid configuration of the feature model and suggest a series of changes to the current invalid configuration to reach the valid configuration. A valid configuration is a labeling of the variables in F (a configuration) such that all of the feature model constraints are satisfied. For each variable f_i, the value should be 1 if the feature is present in the new valid configuration that will be transitioned to. If a feature is not in the new configuration, f_i should equal 0.

We always require $f_1 = 1$ to ensure that the root feature is always selected. For void feature models, there will be no valid solution and the solver will respond that no solution was found. CURE could be used to detect void feature

models but it would be more appropriate to use a technique designed for this purpose, such as [19].

One key input to CURE is the CSP describing the set of all valid feature selections from the feature model (the Feature Model Rules in Table 1). Since these valid feature selections are described as constraints over the variables in F, **a valid labeling of F will always yield a valid feature selection**. Once a valid labeling of F is found, the goal is to determine how to modify the labeling of O to match the valid feature selection denoted by the labeling of F.

First, a constraint must be introduced to model when a feature in the current invalid configuration needs to be deselected to reach the correct configuration. If the i_{th}feature is included in the current configuration ($o_i = 1$), but is not in the new valid configuration ($f_i = 0$), we want the solver to recommend that it be deselected ($d_i = 1$). For every feature, we introduce the following constraint to determine if the i_{th} feature in O needs to be deselected[1]:

$$(f_i = 0) \Rightarrow (o_i = 0 \oplus d_i = 1) \wedge (s_i = 0)$$

If f_i is not selected in the correct configuration ($f_i = 0$), then either the feature was also not selected in the current invalid configuration ($o_i = 0$), or the feature needs to be deselected ($d_i = 1$). Furthermore, if a feature is not needed in the valid configuration ($f_i = 0$) then clearly it should not be a recommended selection ($s_i = 0$).

The solver must also recommend features to select. If the i_{th} feature is selected in the correct and valid configuration $f_i = 1$, and not selected in the current invalid configuration ($o_i = 0$), then it needs to be selected ($s_i = 1$). For each feature, we introduce the constraint:

$$(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0)$$

If a feature is needed by the correct configuration ($f_i = 1$), then either the feature was present in the invalid configuration ($o_i = 1$) or the feature was not present in the invalid configuration and needs to be selected ($s_i = 1$). Clearly, a feature should not be deselected if $f_i = 1$ and thus $d_i = 0$.

The state of each feature, o_i, in the current invalid configuration is compared against the correct state of the feature, f_i, in the valid feature configuration. The behavior of each comparison can fall into four cases:

1. **A feature is selected and does not need to be deselected.** If the i_{th} feature is in the current invalid configuration ($o_i = 1$), and also in the new valid configuration ($f_i = 1$), no changes need be made to it ($s_i = 0$, $d_i = 0$)
2. **A feature is selected and needs to be deselected.** If the i_{th} feature is in the current invalid configuration ($o_i = 1$) but not in the new valid configuration ($f_i = 0$), it must be deselected ($d_i = 1$)
3. **A feature is not selected and does not need to be selected.** If the i_{th} feature is not in the current invalid configuration ($o_i = 0$) and is also not needed in the new configuration ($f_i = 0$) it should remain unchanged ($s_i = 0$, $d_i = 0$)
4. **A feature is not selected and needs to be selected.** If the i_{th} feature is not selected in the current invalid configuration ($o_i = 0$) but is present in the new correct configuration ($f_i = 1$), it must be selected ($s_i = 1$)

[1]The symbol "$\oplus$" denotes *exclusive or*

3.3 Optimal Diagnosis Method

The next step in the CURE diagnosis process is to use the solver to label the variables and produce a series of recommendations. For any given configuration with a conflict, there may be multiple possible ways to eliminate the problem. For example, in the automotive example from Section 2.3, the valid corrective actions were either (1) remove the *1 Mbit/s CAN Bus* and select the *250 Kbit/s CAN Bus* or (1) remove the *Non-ABS Controller* and select the *ABS Controller*. We must therefore tell the solver how to select which of the (many) possible corrective solutions to suggest to developers.

The most basic suggestion selection criteria developers can use to guide the solver's diagnosis is to tell it to minimize the number of changes to make to the current configuration, *i.e.*, prefer suggestions that require changing as few things as possible in the current invalid configuration. To implement this approach, we solve for a CSP labeling that minimizes the sum of variables in $S \cup D$, which is the total number of changes that the solution requires the developer to make. By minizing this sum we therefore minimize the total number of required changes.

Each labeling of the diagnostic CSP will produce two sets of features corresponding to the features that should be selected (S) and deselected (D) to reach the new valid configuration. Developers can ask the solver to cycle through the different potential labelings of the diagnostic CSP to evaluate potential remedies. Furthermore, each new labeling (new diagnosis) also causes the solver to backtrack and create new values for F, which allows developers to evaluate not only the suggested modifications but the configuration that the remedy will produce. Another way to further refine the guidance for the diagnosis is to constrain the new state captured in the labeling of F. This technique is utilized by the extensions in Sections 4.2 and 4.3.

Table 1 shows a complete set of inputs and output suggestions for diagnosing the automotive software example from Section 2.3. If there are multiple labelings of the CSP, initially only one will be returned. After the first solution has been found, however, the solver can much more efficiently cycle through the other equally ranked sets of corrective suggestions.

4 Solution Extensibility and Benefits

This section presents different benefits of CURE and possible ways of extending it.

4.1 Bounding Diagnostic Method

Due to time constraints, it may not be possible to find the optimal number of changes for extremely large feature models. In these cases, a more scalable approach is to attempt to find any suggestion that requires fewer than K changes or with a cost less than K. Rather than directly asking for an optimal answer, we add the following constraint to the CSP and ask the solver for any solution:

$$\sum_{i=1}^{n} s_i + d_i \leq K$$

The sum of all variables $s_i \subset S$ and $d_i \subset D$ represents the total number of feature selections and deselections that need to be made to reach the new valid configuration. Therefore, the sum of both of these sets is the total number of modifications that must be made to the original invalid configuration. The new constraint, ensures that the solver only accepts diagnosis solutions that require the developer to make K or fewer changes to the invalid solution.

The solver is asked for **any** answer that meets the new constraints. In return, the solver will provide a solution that is not necessarily perfect, but which fits our tolerance for change. If no solution is found, we can increment K by a factor and re-invoke the solver or reassess our requirements. As is shown in Section 5.4, searching for a bounded solution rather than an optimal solution is significantly faster.

If the solver cannot find a diagnosis that makes fewer than K modifications, it will state that there is no valid solution that fits a K change budget.

4.2 Debugging from Different Viewpoints

As we discussed in Section 2.3, we need the ability to debug the configuration from different viewpoints. Each viewpoint represents a set of features that the solver should avoid suggesting to add or remove from the current configuration. For example, using the automobile scenario from Section 2.3, the solver can debug the problem from the point of view that hardware decisions trump software by telling the solver not to suggest selecting or deselecting any hardware features.

Debugging from a viewpoint works by pre-assigning values for a subset of the variables in F and O. For example, to force the feature f_i currently in the configuration to remain unaltered by the diagnosis, the values $f_i = 1$ and $o_i = 1$ are provided to the solver. Since $(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0)$, pre-assigning these values will force the solver to label $s_i = 0$ and $d_i = 0$.

To debug from a given point of view, for each feature f_v, in that viewpoint, we first add the constraints, $f_v = 1$, $o_v = 1$, $s_v = 0$, and $d_v = 0$, as shown in Figure 3. The

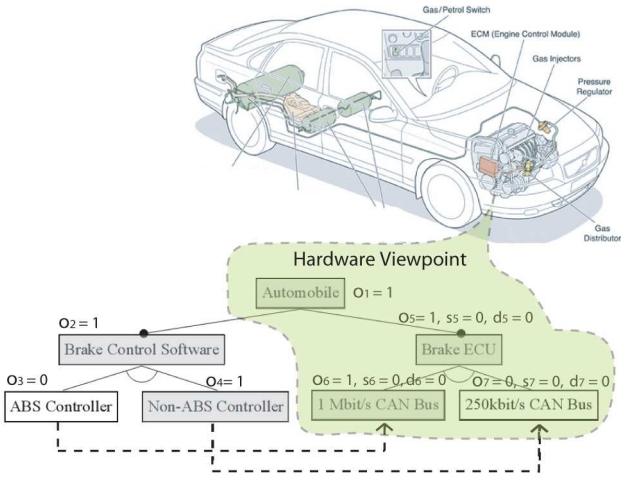

Figure 3: Debugging from a Viewpoint

solver then derives a diagnosis that recommends alterations to other features in the configuration and maintains the state of each feature f_v. The CURE diagnostic model can therefore be used to debug from different viewpoints and address Challenge 3 from Section 2.3.

Pre-assigning values for variables in F and O can also be used to debug staged configuration errors from Challenge 1, Section 2.1. With staged configuration errors, at some point in time T', developers need to select a feature that is in conflict with one or more features selected at time $T < T'$. To debug this type of conflict, developers pre-assign the desired (but currently unselectable) feature at time T' the value of 1 for its o_i and f_i variables. Developers can also pre-assign values for one or more other features decisions from previous stages of the configuration that must not be altered. The solver is then invoked to find a configuration that includes the desired feature at T' and minimizes the number of changes to feature configuration decisions that were made at all points in time $T < T'$.

4.3 Cost Optimal Conflict Resolution

As shown in Section 2.2, conflicts can occur when multiple stakeholders in a configuration process pull the solution in different directions. Debugging tools are therefore needed to mediate the conflict in a cost conscious manner. For example, when a car's software configuration is incompatible with the legacy ECU configuration, it is (probably) cheaper to change the software configuration than to change the ECU configuration and the assembly process of the car. The solver should therefore try to minimize the overall cost of the changes.

We can extend the CSP model to perform cost-based feature selection and deselection optimization. First, we extend the CURE model to associate a cost variable, $b_i \subset B$, with each feature in the feature model. Each cost variable represents how expensive (or conversely how beneficial) it is

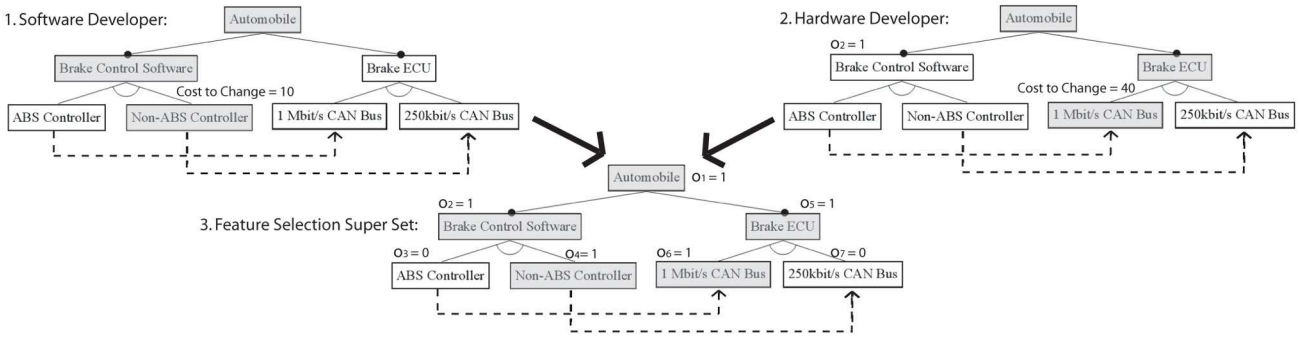

Figure 4: Constructing the Feature Selection Superset for Conflict Mediation

for the solver to recommend that the state of that feature be changed. Before each invocation of the debugger, the stakeholders provide these cost variables to guide the solver in its recommendations of features to select or deselect.

Next, we construct the superset of the features that the various stakeholders desire, as shown in Figure 4. The superset represents the ideal, although incorrect, configuration that the stakeholders would like to have. The goal is to find a way to reach a correct configuration from this superset of features that involves the lowest total cost for changes. The superset is input to the solver as values for the variables in O.

Finally, we alter our original optimization goal so that the solver will attempt to minimize (or maximize) the cost of the features it suggests selecting or deselecting. We define a global cost variable G and let G capture the sum of the costs of the changes that the solver suggests:

$$G = \sum_{i=1}^{n} (d_i * b_i) + (s_i * b_i)$$

G is thus equal to the sum of the costs of all features that the solver either recommends to select or deselect. Rather than instructing the solver to minimize the sum of $S \cup D$, we ask it to minimize or maximize G.

The result of the labeling is a series of changes needed to reach a valid configuration that optimally integrates the desires and decisions of the various stakeholders. Of course, one particular stakeholder may have to incur more cost than another in the interest of reaching a globally better solution. Further constraints, such as limiting the maximum difference between the cost incurred by any two stakeholders, could also be added. The mediation process can be tuned to provide numerous types of behavior by providing different optimization goals. This CSP diagnostic method enables CURE to address Challenge 2 from Section 2.2.

5 Empirical Results

Effective automated diagnostic methods should scale to handle feature models of production systems. This section presents empirical results from experiments we performed to evaluate the scalability of CURE. We compare the scalability of both CURE's optimal and bounding methods from Sections 3.3 and 4.1.

5.1 Experimental Platform

To perform our experiments, we used the implementation of CURE that is provided by the Model Intelligence libraries from the Eclipse Foundation's Generic Eclipse Modeling System (GEMS) project [3]. Internally, the GEMS Model Intelligence implementation of CURE uses the Java Choco Constraint Solver [1] to derive labelings of the diagnostic CSP. The experiments were performed on a computer with an Intel Core DUO 2.4GHZ CPU, 2 gigabytes of memory, Windows XP, and a version 1.6 Java Virtual Machine (JVM). The JVM was run in client mode using a heap size of 40 megabytes (-Xms40m) and a maximum memory size of 256 megabytes (-Xmx256m).

A challenging aspect of the scalability analysis is that CSP-based techniques can vary in solving time based on individual problem characteristics. In theory, CSP's have exponential worst case time complexity, but are often much faster in practice. To evaluate CURE, therefore, it was necessary to apply it to as many models as possible. The key challenge with this approach is that hundreds or thousands of real feature models are not readily available and manually constructing them is impractical.

To provide the large numbers of feature models needed for our experiments, therefore, we built a feature model generator that randomly creates feature models with the desired branching and constraint characteristics. We also imbued the generator with the capability to generate feature selections from a feature model and probabilistically insert a bounded number of errors/conflicts into the configuration. The feature model generator and code for these experiments is available in open-source form from [2].

From preliminary feasibility experiments we conducted, we observed that the branching factor of the tree had little effect on the algorithm's solving time. We also compared diagnosis time using models with 0%, 10%, and 50%

cross-tree constraints and saw that the each increment in the percentage of cross-tree constraints improved performance. For example, with the optimal method and 1,000 feature models, the average diagnosis time gradually decreased from 47 seconds with 0% cross-tree constraints to 36 seconds with 50% cross-tree constraints. The key indicator of the solving complexity was the number of XOR- or cardinality-based feature groups in a model. XOR and cardinality-based feature groups are features that require the set of their selected children to satisfy a cardinality constraint (the constraint is 1..1 for XOR).

For our tests, we limited the branching factor to at most five subfeatures per feature. We also set the probability of XOR- or cardinality-based feature groups being generated to 1/3 at each feature with children. We chose 1/3 since most feature models we have encountered contain more required and optional relationships than XOR- and cardinality-based feature groups. The total number of cross-tree constraints was set at 10%. We also eliminated all diagnosis results from void feature models, since void feature models produced faster diagnostic times and would have skewed the results towards smaller solving times.

To generate feature selections with errors, we used a probability of 1/50 that any particular feature would be configured incorrectly. For each model, we bounded the total errors at 5. In our initial experiments, the solving time was not affected by the number of errors in a given feature model. Again, the prevalence of XOR- or cardinality-based feature groups was the key determiner of solving time.

5.2 Bounding Method Scalability

First, we tested the scalability of the less computationally complex bounding diagnosis method. The speed of the bounding technique allowed us to test 2,000 feature models at each data point (2,000 different variations of each size feature model) and test the bounding method's scalability for feature models up to 500 features. With models above 500 features, we had to reduce the number of samples at each size to 200 models due to time constraints. Although these samples are small, they demonstrate the general performance of our technique. Moreover, the results of our experiments with feature models up to 500 features were nearly identical with sample sizes between 100 and 2,000 models.

Figure 5 shows the time required to diagnose feature models ranging in size from 50 to 500 features using the bounded method. The figure captures the worst and average solving time in the experiments. As seen from the results, our technique could diagnose 500 feature models in an average of ≈300ms.

The upper bound used for this experiment was a maximum of 10% feature selection changes. When the feature bound was too tight for the diagnosis (*i.e.*, more were needed to reach a correct state) the solver quickly declared there was no valid solution. We therefore discarded all instances where the bound was too tight to avoid skewing the results towards shorter solving times.

Figure 5 shows the results of testing the solving time of the bounding method on feature models ranging in size from 500 to 5,000 features.

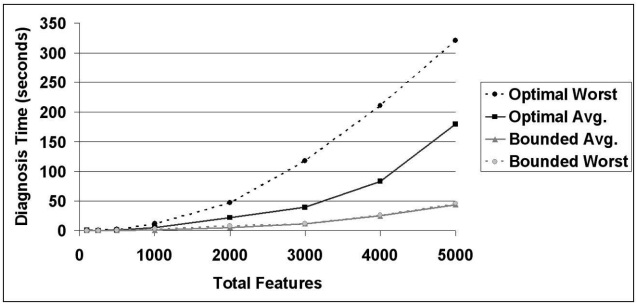

Figure 5: Diagnosis Time for Both Methods for Large Feature Models

Models of this size were sufficient to demonstrate scalability for common production systems. The results show that for a 5,000 feature model, the average diagnosis time was ≈ 50 seconds.

Another key variable we tested was how the tightness of the bound on the maximum number of feature changes affected the solving time of the technique. We took a set of 200 feature models and applied varying bounds to see how the bound tightness affected solution time. Figure 6 shows that tighter bounds produced faster solution times. These results indicate that tighter bounds allow the solver

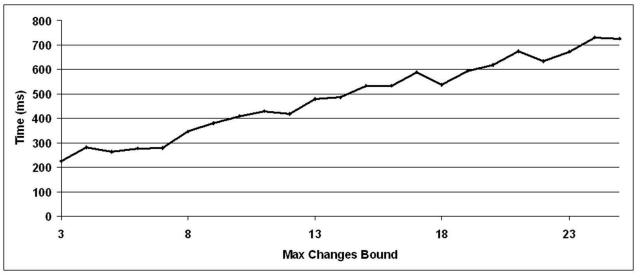

Figure 6: 500 Feature Diagnosis Time with Bounding Method and Varying Bounds

to discard infeasible solutions more quickly and thus arrive at a solution faster.

5.3 Optimal Method Scalability

Next, we tested the scalability of the optimal diagnosis method using 2,000 samples below 500 features and 200 samples for all larger models. Figure 5 shows the results from feature models up to 500 features. At 500 features, the optimal method required an average of ∼1.5 seconds to produce a diagnosis. Figure 5 also shows the tests from

larger models ranging in size up to 5,000 features. For a model with 5,000 features, the solver required an average of ∼3 minutes per diagnosis.

5.4 Comparative Analysis of Optimal and Bounding Methods

Finally, we compared the scalability and quality of results produced with the two methods. Figure 5 shows the bounding method performs and scales significantly better than the optimal method. For feature models of up to 1,000 features, however, both techniques take less than 5 seconds and the optimal method is the better choice. This result raises the question of how much of a tradeoff in solution quality for speed is made when the bounding method is used over the optimal method for larger models.

The bound that is chosen determines the quality of the solution that is produced by the solver. The optimality of a diagnosis given by the bounding method is the number of changes suggested by the bounding method, $Bounded(S \cup D)$, divided by the optimal number of changes, $Opt(S \cup D)$, which yields $\frac{Bounded(S \cup D)}{Opt(S \cup D)}$. Since the bounding method uses the constraint $(S \cup D) \leq K$ to ensure that at most K changes are suggested, we can state the worst case optimality of the bounded method as $\frac{K}{Opt(S \cup D)}$. The closer our bound, K, is to the true optimal number of changes to make, the better the diagnosis will be.

Since tighter bounds produce faster solving times *and* better results, debuggers should start with very small bounds and iteratively increase them upward as needed. One approach is to layer an adaptive algorithm on top of the diagnosis algorithm to move the bound by varying amounts each time the bound proves too tight. Another approach is to employ binary search to hone in on the ideal bound. We will investigate both techniques in future work.

5.5 Debugging Scenarios

Staged configuration and viewpoint debugging (Challenges 1 & 3) are special cases of the technique where the solver is not allowed to modify the selection state of one or more features (*i.e.*, the viewpoint or the feature at time T'). Both of these special cases of debugging actually reduce the search space by fixing values for one or more of the CSP variables. For example, performing staged configuration debugging, which fixes the value for one CSP variable, on a model with 1,000 features, reduced the optimal method's average solving time by $\approx$ 2.5 seconds and the bounding method by $\approx$ .1 seconds.

Cost-based conflict mediation (Challenge 2) performs identically to the standard diagnosis technique. Cost-based mediation merely introduces a series of coefficients, $b_i \subset B$ into the optimization goal. These coefficients do not increase solving time. Furthermore, initiating the diagnosis method with the superset of the configuration participants' desired feature selections also did not impact performance.

6 Related Work

In prior work [19], Trinidad et al. have shown how feature models can be transformed into diagnosis CSPs and used to identify *full mandatory features*, *void features*, and *dead feature models* [19]. Developers can use this diagnostic capability to identify feature models that do not accurately describe their products and to understand why not. The technique we described in this paper builds on this idea of using a CSP for automated diagnosis. Whereas Trinidad focuses on diagnosing feature models that do not describe their products, we build an alternate diagnosis model to identify conflicts in feature configurations. Moreover, we provide specific recommendations as to the minimal set of features that can be selected or deselected to eliminate the error.

Batory et al. [4] also investigated debugging techniques for feature models. Their techniques focus on translating feature models into propositional logic and using SAT solvers to automate configuration and verify correctness of configurations. In general, their work touches on debugging feature models rather than individual configurations. Our approach focuses on another dimension of debugging, the ability to pinpoint errors in individual configurations and to specify the minimal set of feature selections and deselections to remove the error. Furthermore, propositional logic-based approaches do not typically provide maximization or minimization as primitive functions provided by the solver. Since, CURE uses a CSP-based approach, minimization/maximization diagnosis functionality is built-in.

Mannion et al. [15] present a method for encoding feature models as propositional formulas using first-order logic. These propositional formulas can then be used to check the correctness of a configuration. Mannion, however, does not touch on how incorrect configurations are debugged. In contrast, our technique provides this capability and can therefore recommend the minimal feature modifications to rectify the problem.

Pure::variants [8], Feature Modeling Plugin (FMP) [10], FeAture Model Analyser (FAMA) [6], and Big Lever Software Gears [9] are tools developed to help developers create correct configurations of SPL feature models. These tools enforce constraints on modelers as the features are selected. None of these tools, however, addresses cases where feature models with incorrect configurations are created and require debugging. The technique described in this paper provides this missing capability. These tools and our approach are complementary since the tools help to ensure that correct configurations are created and our technique diagnoses incorrect configurations that are built.

7 Concluding Remarks

It is hard to debug conflicts and errors in large feature models created through staged or multi-stakeholder configuration [5]. This paper described a technique, called CURE,

that uses CSPs to diagnose errors and conflicts in configurations. CURE's diagnoses can specifically recommend the minimum or cost optimal set of features that should be selected or deselected in a faulty configuration.

CURE's CSP-based diagnosis model is extensible and can be modified to perform conflict mediation, run faster, or debug from different viewpoints. Moreover, empirical results show CURE can scale to production feature models with 5,000 features and still provide a diagnosis in between 45 seconds and 4 minutes. These time bounds should be sufficient for the design-time use of this algorithm.

The following are lessons learned from our efforts:

- CURE can scale to feature models with several thousand features.
- The optimality of the diagnosis provided by the bounding method is determined by how close K is set to the true minimum number of features that need to be changed to reach a valid state. Setting an accurate bound for K is not easy. In future work, we plan to investigate different methods of honing the boundary used in the bounding method.
- The same CSP can often be stated in multiple ways. Different formulations can yield different performance characteristics. In future work, we intend to see if it is possible to vary the diagnosis CSP formulation and show that the technique can scale to even larger models while still providing reasonable runtimes.

The diagnosis technique has been implemented as part of the GEMS EMF Intelligence project and is available from `www.eclipse.org/gmt/gems`. We intend to integrate this technique into the FAMA tool [6] as well.

References

[1] Choco constraint programming system. http://choco.sourceforge.net/.

[2] Experimental platform, http://www.dre.vanderbilt.edu/~jules/splc08.zip.

[3] Generic eclipse modeling system (gems) http://eclipse.org/gmt/gems.

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

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

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

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

[8] D. Beuche. Variant Management with Pure:: variants. Technical report, Pure-Systems GmbH, http://www.pure-systems.com, 2003.

[9] R. Buhrdorf, D. Churchett, and C. Krueger. Salion's Experience with a Reactive Software Product Line Approach. In *Proceedings of the 5th International Workshop on Product Family Engineering*, Siena, Italy, November 2003.

[10] K. Czarnecki, M. Antkiewicz, C. Kim, S. Lau, and K. Pietroszek. In *FMP and FMP2RSM: Eclipse Plug-ins for Modeling Features Using Model Templates*, pages 200–201. ACM Press New York, NY, USA, October 2005.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004: Proceedings*, 2004.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005.

[13] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *constraints*, 2(2):0.

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

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

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

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

[18] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[19] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software, in press*, 2007.

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

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