Automated Diagnosis of Feature Model Configurations

J. White*[,a], D. Benavides[b], D.C. Schmidt[a], P. Trinidad[b], B. Dougherty[a], A. Ruiz-Cortes[b]

[a]*Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37204, USA*
[b]*Dept. of Computer Languages and Systems University of Seville, Avda. de la Reina Mercedes, s/n B-41012 Seville, Spain*

Abstract

Software product-lines (SPLs) are software platforms that can be readily reconfigured for different project requirements. A key part of an SPL is a model that captures the rules for reconfiguring the software. SPLs commonly use feature models to capture SPL configuration rules. Each SPL configuration is represented as a selection of features from the feature model. Invalid SPL configurations can be created due to feature conflicts introduced via staged or parallel configuration or changes to the constraints in a feature model. When invalid configurations are created, a method is needed to automate the diagnosis of the errors and repair the feature selections.

This paper provides two contributions to research on automated configuration of SPLs. First, it shows how configurations and feature models can be transformed into constraint satisfaction problems to automatically diagnose errors and repair invalid feature selections. Second, it presents empirical results from diagnosing configuration errors in feature models ranging in size from 100 to 5,000 features. The results of our experiments show that our CSP-based diagnostic technique can scale up to models with thousands of features.

Key words: software product-lines, configuration, diagnosis, constraint satisfiaction, optimization

1. Introduction

Background and challenges. Software product-lines (SPLs) are software platforms that can be reconfigured for different requirement sets (Clements and Northrop, 2002). SPLs are designed to facilitate the reuse of software assets across multiple projects and amortize development cost. 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, thus allowing it to be used in cars aimed at different price points. Each configuration of an SPL is called a *variant*.

A core component of an SPL is a model of the points of variability within the software architecture. These variability points determine how the software can be reconfigured to meet new requirement sets. Points of variability are typically not independent of one another, however, and thus an explicit set of rules must be developed to dictate legal combinations of configuration settings *e.g.*, a car cannot have both ABS and standard braking software controllers.

Feature modeling (Kang et al., 1998) is a technique for documenting the points of variability in an SPL, how the points of variability affect one another, and what constitutes a complete configuration of the SPL. A feature model leverages *features* as the abstraction for documenting configuration rules. Each feature represents an incre-

*Corresponding author
Email addresses: jules@dre.vanderbilt.edu (J. White), benavides@us.es (D. Benavides), schmidt@dre.vanderbilt.edu (D.C. Schmidt), ptrinidad@us.es (P. Trinidad), briand@dre.vanderbilt.edu (B. Dougherty), aruiz@us.es (A. Ruiz-Cortes)

ment in product functionality. A feature model can capture different types of variability, ranging from ranging from requirements variability to implementation variability (Metzger et al., 2007).

Unique configurations of an SPL are captured as selections of features from a feature model. 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 SPL research has focused on ensuring that features chosen from feature models are correct and consistent with the SPL and variant requirements. For example, boolean circuit satisfiability techniques (Mannion, 2002) and Constraint Satisfaction Problems (CSPs) (Benavides et al., 2005; White et al., 2007) have been used 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 (Buhrdorf et al., 2003), Pure::variants (Beuche, 2003), FeAture Model Analyser (FAMA) (Benavides et al., 2007), and the Feature Model Plug-in (Czarnecki et al., 2005a), to support the construction of feature models and correct selection of feature configurations.

Although SPL developers are equipped with a number of tools and techniques, configuration problems may occur due to human errors, such as miscommunication between engineers, or other factors, such as requirement changes. For example, configurations of large feature models may be produced using *staged configuration* (Czarnecki et al., 2004, 2005b), where features are iteratively selected by a group of participants. At each stage, one participant chooses features before passing the partial configuration on to the participant in the next stage. In these staged configuration processes, a participant may select a feature that conflicts with a critical feature needed by a participant in a later stage. Due to the complexity of feature model constraints it may be hard to forsee these conflicts. It is also hard to debug a configuration to figure out how to change decisions in previous stages to make the critical feature selectable (Batory et al., 2006).

In many cases—particularly when SPLs are produced from supply-chains—the configuration decisions of multiple developers working in parallel must be integrated. Since developers may work in different departments, organizations, or geographic locations where they are not in continuous direct communication or have conflicting goals, configurations may be produced with conflicting feature selections. For example, hardware developers of one subcontractor for an automobile may desire a lower cost set of Electronic Control Units (ECUs) that cannot support the embedded controller code features needed by the software developers of another subcontractor. Even though the two teams may share the same toolset, because the tools cannot prevent errors in a distributed environment where the teams are not in continuous direct communication, conflicts may arise. Methods are therefore needed to (1) evaluate and debug conflicts between participants and (2) recommend modifications to the participants feature selections to make them compatible.

A further challenge to maintaining error free SPL configurations is that external influences (such as supply-chain disruption, changes in market demand, or safety regulations) may change over time and thus invalidate previously correct configurations. For example, if a new government regulation is passed that mandates higher emissions standards, it may necessitate changing existing software configurations. A critical question is thus often determining the lowest cost method of updating existing software configurations to meet the new external regulations and requirements, which is a process termed *software configuration evolution*.

Although prior research has shown how to identify flawed configurations (Batory, 2005; Mannion, 2002), conventional debugging mechanisms cannot pinpoint configuration errors and identifying corrective actions. More specifically, tech-

niques are needed that can take an arbitrary flawed configuration and produce the minimal set of feature selections and deselections to bring the configuration to a state that satisfies all of the feature model rules. This paper focuses on addressing these gaps in existing research. The solutions that the paper presents to these gaps are described in Section 2.

Solution overview and contributions. Our approach to debugging feature model configurations and automating configuration evolution transforms an invalid feature model configuration into a Constraint Satisfaction Problem (CSP) (Van Hentenryck, 1989) 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 the *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 that can pinpoint conflicts and constraint violations in feature models
2. We provide a simplified diagnostic CSP that is not as flexible as the full diagnostic CSP but reduces diagnosis time
3. We show how CURE can remedy a configuration error by automatically deriving the minimal set of features to select and deselect
4. We provide mechanisms for using CURE to mediate conflicting configuration participant feature selection desires via cost optimization techniques
5. We show how CURE allows stakeholders to debug a configuration error or conflict from different viewpoints
6. We show how CURE can be used to automate feature selection evolution
7. We provide empirical results showing that CURE's scalability can support industrial SPL feature models containing over 5,000 features.
8. We present empirical results showing the improvement in solving time that the simplified CSP provides

Our prior work on automated feature configuration (Trinidad et al., 2007) focused on detecting the most common types of errors in the constraints of a feature model, such as *void features*, using pre-defined configurations. In further research (White et al., 2008), we developed new techniques for diagnosing the configurations of feature models rather than the feature model itself. In this paper, we also investigate feature expansion planning techniques and adapting to supply-chain disruption in the context of feature models. Furthermore, the prior work did not account for how manual configuration decisions that led to the inability of satisfying project requirements were handled. Batory et al. (Batory et al., 2006) have described this is an important challenge in the field automated software product-line configuration. In this work, we address this gap in our prior work by showing how automated remediation can be performed to eliminate configuration conflicts that arise from human error or concurrent configuration processes.

This paper extends our previous work on automated diagnosis of product-line configuration errors (White et al., 2008) in three ways. First, we provide a new methodology for applying our CURE diagnosis technique to automating the evolution of feature model configurations, which allows SPL configurations to evolved automatically over time to meet changing external requirement sets. Second, we have significantly expanded the discussion of CURE's relationship with related work on SPL configuration.

Third, in some scenarios, developers may desire to tradeoff diagnosis flexibility for improved solving performance. For example, developers may wish to eliminate diagnostic capabilities that they do not use in exchange for faster diagnostics for large feature models. We provide a new simplified diagnostic CSP formulation that can significantly reduce solving time for large feature models and present new empirical results demonstrating the improvement in solving time provided by the new CSP formulation. For all the techniques proposed in this paper, we make the implicit assumption that the constraints in the feature model itself are correct. The process described by Trinidad

et al. (Trinidad et al., 2007) can be used to determine if the feature model is error free and suitable for the techniques described in this paper.

Paper organization. The remainder of the paper is organized as follows: Section 2 presents the CURE CSP-based technique for diagnosing configuration errors and conflicts; Section 3 describes CURE CSP modifications that can be made to decrease solving time; Section 4 shows how CURE can be used to perform conflict mediation, multi-viewpoint debugging, and staged configuration debugging; Section 5 shows how CURE can be used to automate software feature configuration evolution decisions; Section 6 presents empirical results demonstrating the ability of CURE to scale to feature models with thousands of features; Section 7 compares CURE with related research; and Section 8 presents concluding remarks.

2. Configuration Error Diagnosis with CURE

Configuration Understanding and REmedy (CURE) is a constraint-based technique for diagnosing errors and mediating conflicts in feature model configurations. Developers can use CURE to identify the minimal set of features needed to select or deselect to transform an invalid input 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. (Benavides et al., 2005) have shown how feature models can be transformed into CSPs to automate feature selection with a constraint solver (Jaffar and Maher, 1994). Trinidad et al. (Trinidad et al., 2007) 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 (Reiter, 1987). This section presents an alternative CSP diagnostic model, not related to Reiter's theory of diagnosis, for deriving the minimum set of features that should be selected or deselected to eliminate a conflict in a feature configuration.

CURE can diagnose feature models that use standard feature modeling notations, such as required features, XOR groups, and optional features. It is possible to also encode cardinality groups into the diagnostic CSPs used by CURE, but we do not cover this in the paper. Feature quality attributes can also be taken into account in the diagnostic process, and we discuss cost-based quality attributes in Section 4.2.

2.1. 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. A constraint solver finds a valid *labeling* (set of variable values) that simultaneously satisfies all constraints in the CSP. A valid labeling of the CSP would be ($A = 1$, $B = 2$).

To build a CSP to model a feature model configuration problem, 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 $f_i \in F$, $f_i = 1$ 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 with respect to the feature model rules. 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. More detailed descriptions of the steps for transforming a feature model to a CSP appear in (Benavides et al., 2005).

2.2. Configuration Diagnostic CSP

This section describes CURE's diagnostic CSP. First, we introduce the diagnostic recommendation variables that CURE uses to indicate which features should be selected or deselected to fix a configuration. Next, we describe how the diagnosis of a flawed configuration is modeled as a CSP using these variables. Finally, we explain how the constraints on the diagnostic variables produce a correct diagnosis of an invalid configuration.

2.2.1. Diagnostic Recommendation Variables

As a motivating example, we use a feature model for an automobile. Figure 1 shows a simple feature model for an automobile. Assume that an invalid configuration with the features `Automobile`, `Brake Control Software`, `Non-ABS Controller`, `Brake ECU`, and `1 Mbit/s CAN Bus` has been created.

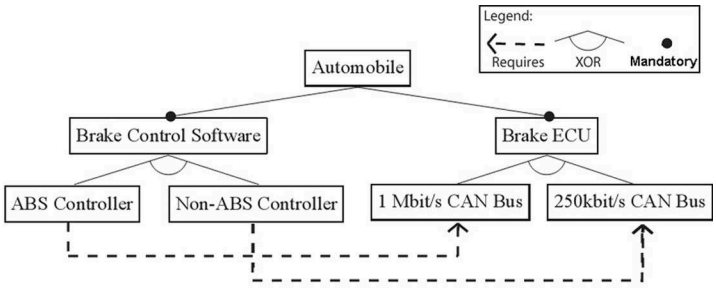

Figure 1: Simple Feature Model for an Automobile

To diagnosis this flawed configuration, developers need a list of features that should be selected or deselected to make the invalid configuration a valid configuration. CURE takes an invalid configuration and a set of constraints describing a feature model as input and produces a valid configuration along with the features to select and deselect to reach the valid output configuration from the input configuration.

The invalid input configuration is provided to CURE as a set of labeled variables, O. In turn, CURE produces an output configuration as a set of labeled variables F, that describe a valid configuration that can be reached by modifying the input configuration O. Since the goal is to produce a valid output configuration, the feature model constraints are input into CURE as constraints on the output variables F. That is, the feature model constraints define the rules that the correct output configuration must conform to. Finally, CURE also outputs the features that should be selected, S, and deselected, D, to modify the invalid input configuration, O, to match the valid output configuration, F.

The overall diagnostic process for CURE is shown in Figure 2.

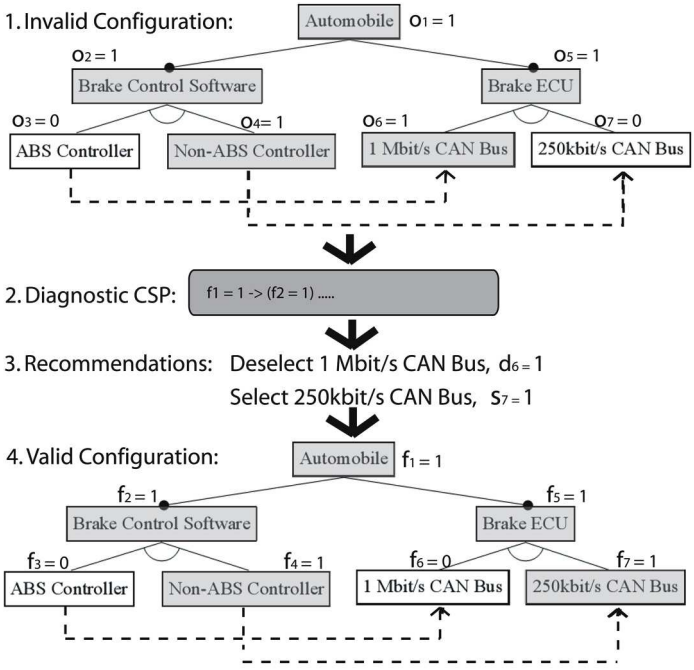

Figure 2: Diagnostic Technique Architecture for CURE

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

ration to create a valid configuration where each variable f_i equals 1 if the corresponding feature is selected in the new and valid output configuration. For example, $f_7 = 1$ means that the *250 Kbit/s CAN Bus* is selected in the new valid configuration.

The variables, $o_i \in O$, are used to input the current invalid configuration of the feature model into CURE. If the i_{th} feature is currently selected in the invalid configuration, $o_i = 1$. If the feature is not selected, $o_i = 0$. An SPL engineer provides these values to CURE as the input for the algorithm.

To enable the constraint solver to recommend features to select and deselect, two sets of output 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 $s_i \in S$ indicates that the i_{th} feature should be added as a selected feature to the invalid input configuration captured in O. Similarly, $d_i = 1$ implies that the feature i_{th} should no longer be a selected feature in the invalid input configuration.

The other output of CURE is the actual feature selection that will be reached by applying the recommended feature selection changes to the invalid input configuration. The variables, $f_i \in F$, specify the new feature selection that is reached by applying the S and D recommendations to the variables in O. That is, the F variables contain the new feature selection that CURE recommends the SPL engineers should use in place of the invalid configuration. Table 1 lists the complete set of diagnostic variables that would be used by CURE for the feature model in Figure 1.

Variables	
Variable Explanations	$f_i \in F$: feature variables for the valid configuration that will be output by CURE; $o_i \in O$: the features selected in the invalid input configuration; $s_i \in S$: features to select to reach the valid configuration; $d_i \in D$: features to deselect 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:	$\forall f_i \in F$ $\{$ $(f_i = 1) \Rightarrow (o_i = 1 \oplus s_i = 1) \wedge (d_i = 0)$ $(f_i = 0) \Rightarrow (o_i = 0 \oplus d_i = 1) \wedge (s_i = 0)$ $\}$
Outputs	
Features to Select:	$s_1 = 0, s_2 = 0, s_3 = 0, s_4 = 0, s_5 = 0, s_6 = 0,$ $\mathbf{s_7 = 1}$
Features to Deselect:	$d_1 = 0, d_2 = 0, d_3 = 0, d_4 = 0, d_5 = 0,$ $\mathbf{d_6 = 1}$, $d_7 = 0$
New Valid Conf.:	$f_1 = 1, f_2 = 1, f_3 = 0, f_4 = 1, f_5 = 1, f_6 = 0, f_7 = 1$

Table 1: Diagnostic CSP Construction

2.2.2. Diagnostic CSP Constraints

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 invalid input configuration to reach the valid configuration. A valid output 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 output configuration, f_i should equal 0.

We always require $f_1 = 1$ to ensure that the root feature is always selected in the output configuration. For void feature models, it will be im-

possible for the root feature to be selected in the output configuration and there will be no valid solution. In these cases, the solver will respond that no solution was found. CURE could also be used to detect void feature models but it would be more appropriate to use a technique designed for this purpose (Trinidad et al., 2007).

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 invalid input configuration needs to be deselected to reach the correct output configuration. If the i_{th}feature is included in the invalid input configuration ($o_i = 1$), but is not in the output 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 output configuration ($f_i = 0$), then either the feature was also not selected in the invalid input configuration ($o_i = 0$), or the feature needs to be deselected ($d_i = 1$). Furthermore, if a feature is not needed in the output 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 output configuration $f_i = 1$, and not selected in the invalid input 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 output 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 $d_i = 0$.

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

The state of each feature, o_i, in the invalid input configuration is compared against the correct state of the feature, f_i, in the output 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 invalid input configuration ($o_i = 1$), and also in the output 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 invalid input configuration ($o_i = 1$) but not in the output 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 invalid input configuration ($o_i = 0$) and is also not needed in the output 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 invalid input configuration ($o_i = 0$) but is present in the output configuration ($f_i = 1$), it must be selected ($s_i = 1$)

2.3. 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. 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 invalid input configuration. To implement this approach, we solve for a CSP labeling that minimizes:

$$\sum_{0}^{n} s_i + \sum_{0}^{n} d_i$$

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. Moreover, each new labeling (new diagnosis) 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 4.1 and 4.2.

Table 1 shows a complete set of inputs and output suggestions for diagnosing the automotive software example from Section 2.2.1. 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.

3. CURE Performance Enhancements

This section describes a number of CSP modifications that can be used to decrease the diagnostic time of CURE.

3.1. Simplified CSP Formulation

To improve the diagnosis speed of the process, the diagnostic CSP can be modified to include fewer variables, which decreases the flexibility of the types diagnostic constraints and goals that can be utilized, but increases performance. As part of the diagnosis process, the solver must calculate values for the D and S variables, which determine which features to select and deselect. The process of determining whether to select or deselect a feature can easily be performed in code outside of the solver. For example, after the solver has determined the closest valid configuration, Java code can be used to compare it to the invalid configuration and recommend which features to select or deselect via a simple difference of the o_i and f_i variables. Moving these variables out of the diagnostic CSP can improve performance, but eliminates the ability to define constraints or gaols based on the selection and deselection of features.

To simplify the CURE CSP, the D and S variables can be replaced with a single set of variables, N, noting the feature selections that differ between the invalid and closest valid configuration. The N variables do not indicate whether to select or deselect a feature, but merely that its selection state differs in the two configurations. In the simplified CSP formulation, for each feature in the feature model, a corresponding n_i variable is created. Each variable $n_i \in N$ can have value zero or one. Moreover,

$$(n_i = 1) \Rightarrow (o_i \neq f_i)$$

, *i.e.*, n_i will only have value 1 if the selection state of the corresponding feature differs in the invalid and closest valid configurations.

Removing the S and D variables has an impact on the expressiveness of the function that can be used to define what constitutes the closest valid configuration. For example, if we define the optimization goal, G, as the function:

$$G = \sum_{0}^{n} s_i + \sum_{0}^{n} 2d_i$$

the solver can be instructed to preference selecting rather than deselecting features to fix the configuration. That is, the solver will attempt to select new features wherever possible rather than deselect features that are already in the configuration. Likewise, the function tells the solver that adding 3 features is more expensive than deselecting 1. Developers can therefore exercise fine-grained control over the weighting of the selection and deselection of features when diagnosing an invalid configuration.

By removing the D and S variables, the simplified CSP formulation eliminates the ability to put

weights on the relative importance of feature selection and deselection in the diagnosis. As shown by the results in Section 6.5, this tradeoff can yield a significant improvement in diagnosis speed. The simplified CSP formulation is preferable, therefore, unless fine-grained weighting of feature selection and deselection is needed.

3.2. 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 constraint:

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

to the CSP and ask the solver for any solution.

The sum of all variables $s_i \in S$ and $d_i \in D$ represents the total number of feature selections and deselections that need to be made to reach the new valid configuration. The sum of both of these sets is thus 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 optimal, but which fits our tolerance for change. If no solution is found, we can increment K by a factor and reinvoke the solver or reassess our requirements. The results in Section 6.4 show that it is significantly faster to search for a bounded solution rather than an optimal solution.

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. Applying CURE to Industrial Feature Configuration Diagnostic Problems

4.1. Debugging from Different Viewpoints

The feature labeled as the source of an error in a feature model configuration may vary depending on the viewpoint used to debug it. For example, if a configuration in the feature model shown in Figure 1 is created to include 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 software features trump ECU hardware decisions, then the *1 Mbit/s CAN Bus* feature is the error. If we assume that ECU decisions precede software features, 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.

CURE can be provided with aditional constraints to allow for viewpoint-based diagnostics. 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 shown in Figure 1, the solver can debug the problem from the point of view that hardware decisions trump software features by telling the solver not to suggest selecting or deselecting any hardware features.

Debugging from a viewpoint in CURE 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 $d_v = 0$.The solver then derives a diagnosis that recommends alterations to other features in the configuration and maintains the state of each feature f_v.

4.2. Cost Optimal Conflict Resolution

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, $b_i \in B$, with each feature in the feature model. Each cost variable represents how expensive (or conversely how beneficial) it is for the solver to recommend 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 3. 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, as follows:

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

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 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.

4.3. Staged Configuration Conflict Resolution

Another type of conflict can occur in staged configuration (Czarnecki et al., 2004). Staged configuration is a configuration process where developers iteratively select features to reduce the variability in a feature model until a variant is constructed. The need for staged configuration has been demonstrated (Czarnecki et al., 2004, 2005b) in the context of software supply chains for embedded automotive software. 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 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 may conflict with decisions that a stakeholder desires to make at a point 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'.

Pre-assigning values for variables in F and O can also be used to debug staged configuration errors. With staged configuration errors, at some point in time T', developers need to select a feature that is in conflict with one or more fea-

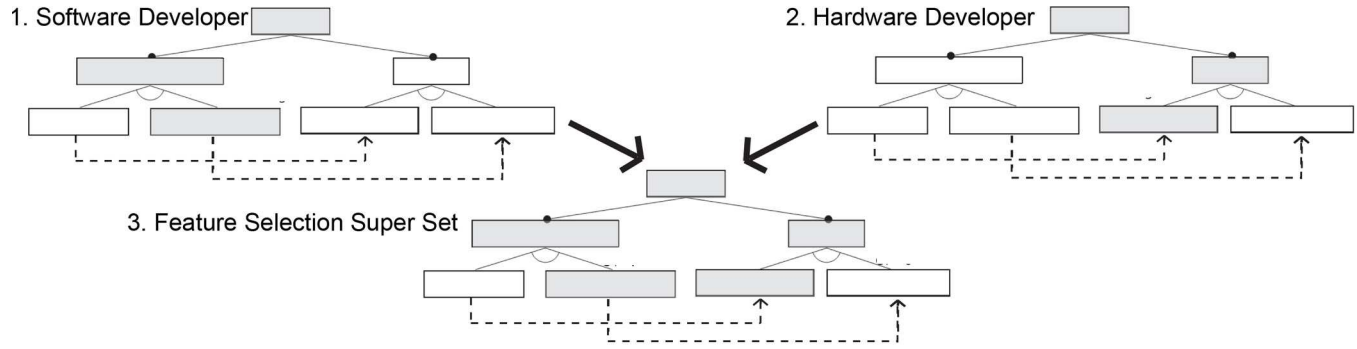

Figure 3: Constructing the Feature Selection Superset for Conflict Mediation

tures 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 $T < T'$.

5. Using CURE for Software Configuration Evolution

Software feature configuration evolution is a critical part of SPL management that involves modifying an existing software configuration to meet new requirements. Although building a variant for a new requirement set is a standard feature model configuration problem, it can not always be performed without taking into consideration existing configurations. For example, if a key supply-chain vendor for an automobile goes out of business, automotive manufacturers must account for the configuration of existing model year cars when determining how to produce new configurations that eliminate the parts produced by the defunct vendor. Simply producing new configurations with no regard for the existing manufacturing, procurement, and other processes investments would be costly and error-prone.

The goal of feature configuration evolution is to transform a current configuration A that satisfies a requirement set R into a new configuration A' that satisfies an evolved requirement set R'. Moreover, because there may be a substantial existing investment in A, a goal of the evolution process is to find A', such that it minimizes or maximizes a function, $\Delta(A, A')$. For example, developers may wish to choose evolution paths that require adding as little additional cost to A as possible and thus $\Delta(A, A')$ computes the difference in cost between the legacy and evolved configurations.

A key difference between configuration diagnosis and configuration evolution is that both A and A' are valid feature model configurations. Configuration diagnosis takes an invalid configuration O and finds a valid configuration F that is as close as possible. Although CURE was originally developed for diagnosing errors in configurations, this section shows that it can also be applied to software configuration evolution problems where the goal is to find A' while optimizing a function $\Delta(A, A')$. This section also presents methods for using CURE to evolve feature model configurations to handle a number of SPL scenarios (such as supply-chain disruption and feature expansion) that necessitate configuration evolution.

5.1. Supply-chain Disruption

The features that are present in an SPL's feature model are dependent on an organization's ability to procure specific sets of software and hardware components. For example, the production of an automobile may depend on vendors providing the ECU and associated software for a high-end infotainment system. If any vendors go out of business, decide to no longer produce one

of the requisite parts, or cannot produce sufficient quantities of a part, SPL developers must determine how to evolve the automobile's configuration to eliminate the unavailable part.

When a part becomes unavailable, developers first need to determine which existing features or combinations of features in the feature model are no longer viable selections due to the unavailable parts. In a simple scenario, a single feature may no longer be available. For example, in the feature model shown in Figure 5, if the ECU required for the *Advanced Infotainment* feature becomes unavailable, this feature will be invalid in a feature selection.

In more complicated situations, alternative parts may be available from other vendors, but these new parts may have different requirements than the old parts. For example, in the scenario depicted in Figure 4 the ECU used for an automobile's infotainment system becomes unavailable and an alternative ECU must be used. The

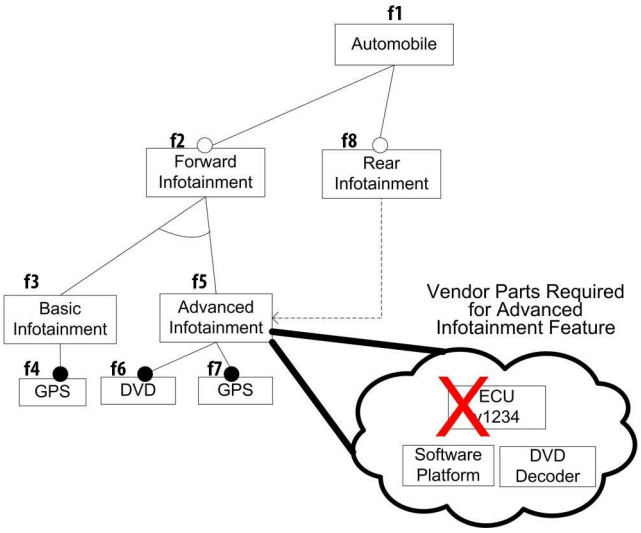

Figure 5: Supply-chain Disruption Leading to Feature Unavailability

new ECU has more substantial peak power requirements, however, which requires adding a new relationship to the feature model. When supply-chain disruption occurs, developers must determine cost-effective ways of evolving legacy configurations to meet the new requirements.

CURE can be used to automate the process of fixing configurations that have become invalid due to supply-chain disruptions. CURE's automated diagnosis abilities allow developers to analyze evolution alternatives that optimize specific properties, such as cost. The key to CURE's ability to automate evolution is that the legacy software configuration, A, can be modeled by the invalid configuration O and the target evolved configuration, A', can be modeled as the closest valid configuration F.

Cure's goal function, G, which is typically a function of O and F, can also be used to model $\Delta(A, A')$ by defining $\Delta(A, A')$ in terms of the O, F, S, and D variables. Thus, we model the evolution process as:

$$C = Feature\ Model\ Constraints$$

$$O = A$$

$$F = A'$$

$$G = \Delta(A, A')$$

$$R = C$$

$$R' = C \wedge Additional\ CSP\ Constraints$$

For example, the *Advanced Infotainment* feature, f_5, in Figure 5 is no longer a valid feature selection because a required ECU is not available. To automate the evolution of each configuration that has this feature selected, CURE can be used to derive the closest valid configuration for which:

$$R' = C \wedge (f_5 = 0)$$

That is, find a feature model configuration which satisfies the feature model constraints, C, and also does not include f_5 in the feature selection $f_5 = 0$. Moreover, designers can use the cost coefficients introduced in Section 4.2 to represent the relative prices of different evolution options and use CURE to perform cost optimal evolution automation by making G calculate the difference in cost between the legacy and evolved configuration.

The more complex evolution scenario outlined in Figure 4 can also be automated with CURE. In this case, developers need to evolve any configurations that include both the *Advanced Infotainment* and *50W* features. Using CURE, developers add the following additional constraint to the

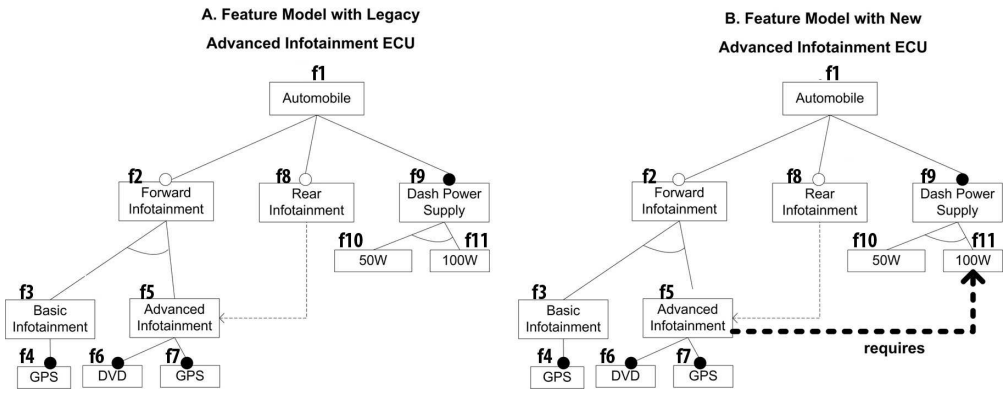

Figure 4: Supply-chain Disruption Leading to Feature Model Rule Changes

diagnostic CSP:

$$R' = C \wedge ((f_5 = 1) \Rightarrow (f_{10} = 1))$$

This additional constraint forces the CSP solver to ensure that any evolved configurations that it suggests adhere to the new feature model rule. Again, developers can apply cost coefficients or constraints requiring that *Advanced Infotainment* be selected in evolved configurations ($f_5 = 1$).

5.2. Feature Expansion Planning

Another important evolution problem that should have automation support is planning feature expansion. Feature expansion planning is determining how to incorporate new features to satisfy new requirements, such as changing market demands, into existing configurations. For example, automotive designers may want to incorporate automated parking assistance into the next year's model of a car. As with supply-chain disruption, developers cannot simply ignore the existing configuration of the current year's model. Instead, developers need to determine the minimal or lowest cost set of modifications that can be applied to the current configuration to enable the newly required features.

To automate feature expansion planning with CURE, the following new requirement set

$$R' = C \wedge (\forall f_i \in New,\ f_i = 1)$$

is produced that requires that each desired new feature, $f_i \in New$, is selected, *i.e.*, the f_i variable for each desired feature, regardless if it is optional, required, or alternative, must have value 1, meaning that the feature is selected. If there are hard requirements on feature selections that can not be altered, these are also encoded into R' as:

$$R' = C \wedge (\forall f_i \in New,\ f_i = 1) \wedge (Hard\ Requirements)$$

For example,

$$R' = C \wedge (f_8 = 1) \wedge (s_2 = 0 \wedge d_2 = 0)$$

finds a configuration in Figure 5 that adds the *Rear Infotainment* feature without modifying the *Forward Infotainment* feature. The constraint ($f_8 = 1$) requires that the *Rear Infotainment* feature be selected by the CSP solver and the constraint ($s_2 = 0 \wedge d_2 = 0$) ensures that no changes are recommended for the *Forward Infotainment* feature. As with previous examples, cost coefficients and the goal function G can be defined to preference the lowest cost evolved configuration.

6. Runtime Evaluation with Randomized Models

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 2.3 and 3.2. Although we assume that the technique is exponential in time complexity, the experiments are designed to demonstrate that the technique is still scalable enough to support industrial feature models with 5,000 features.

6.1. Experimental Platform

To perform our experiments, we used the implementation of CURE provided by the Model Intelligence libraries from the Eclipse Foundation's Generic Eclipse Modeling System (GEMS) project (`eclipse.org/gmt/gems`). Internally, the GEMS Model Intelligence implementation of CURE uses the Java Choco Constraint Solver (`http://choco.sourceforge.net`) 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 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 it was therefore 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 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 (`code.google.com/p/ascent-design-studio`).

From our preliminary feasibility experiments, 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 observed that 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.[2]

Our tests 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.

6.2. Experiment 1: Bounding Method Scalability

Hypothesis. We hypothesized that CURE's CSP-based diagnostic technique could be used to diagnose configurations with 1,000s of features when the bounding method was employed.

Experiment design. To test the diagnostic capabilities of the bounding method, we generated large numbers of feature models and a flawed configuration for each model. The CURE bounded diagnostic method was used to diagnose each of the flawed configurations. The speed of the bounding technique allowed us to test 2,000 feature models at each data point (2,000 different

[2]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).

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.

Analysis of results. Figure 6 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 aver-

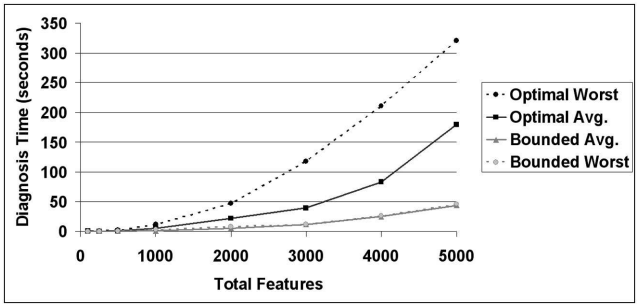

Figure 6: Diagnosis Time for Both Methods for Large Feature Models

age solving time in the experiments. As seen from the results, our technique could diagnose models with 500 features 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 6 shows the results of testing the solving time of the bounding method on feature models ranging in size from 500 to 5,000 features. 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 7 shows that tighter bounds produced faster solution times. These results indicate that tighter

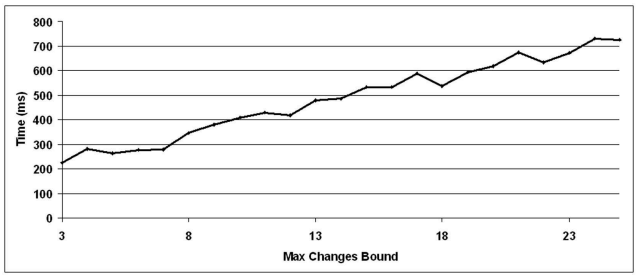

Figure 7: 500 Feature Diagnosis Time with Bounding Method and Varying Bounds

bounds allow the solver to discard infeasible solutions quickly and thus arrive at a solution faster.

6.3. Experiment 2: Optimal Method Scalability

Hypothesis. We hypothesized that the optimal CURE diagnostic method could also scale to feature models with 1,000s of features. We expected that the diagnostic time would be substantially longer than with the bounding method.

Experiment design. We tested the scalability of the optimal diagnosis method using 2,000 samples below 500 features and 200 samples for all larger models. For each feature model, an invalid configuration was generated and diagnosed with CURE. We tracked the best, average, and worst diagnosis time for each size model.

Analysis of results. Figure 6 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 6 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.

6.4. Comparative Analysis of Bounding and Optimal Methods

Finally, we compared the scalability and quality of results produced with the two methods. Figure 6 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.

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 home in on the ideal bound. We will investigate both techniques in our future work.

6.5. Experiment 3: Simplified CSP Formulation

Hypothesis. We hypothesized that the simplified CSP formulation would provide a constant reduction in solving time compared to the standard complex CSP formulation.

Experiment design. We performed experiments to compare the diagnosis speed of the standard complex CSP formulation against the simplified CSP formulation, as described in Section 3.1. We used our feature model generation and configuration infrastructure to produce feature models and flawed configurations of varying sizes and then diagnosed them with both techniques. The results in this section present the differences in diagnosis times that we observed. We generated and solved feature models ranging in size from 500 to 5,000 features. For each size feature model, we solved 20 instances and tracked the average solving time of the two CSP formulations.

Analysis of results. The results are shown in Figure 8. The results in this figure show the simplified CSP technique had a much faster average diagnosis time. For example, with 5,000 features, the simplified CSP formulation had an average diagnosis time of less than half of the standard complex CSP formulation.

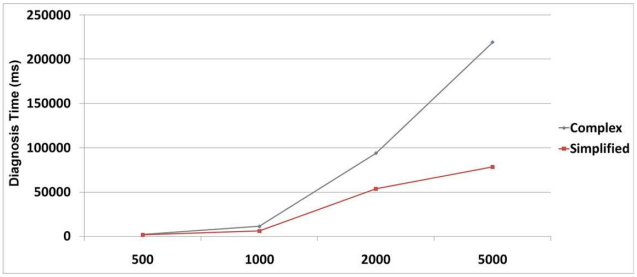

Figure 8: Comparison of Diagnosis Times for the Complex and Simplified CSP Formulations on Feature Models with 500 to 5,000 Features

We took the data shown in Figure 8 and analyzed it to calculate the percentage of reduction in average solving time provided by the simplified CSP formulation over the complex formulation. Figure 9 shows the results of this analysis. For smaller feature models with 500 features, the simplified technique provided a 25% reduction in diagnostic time. As the size of the feature models grew, moreover, the percentage of reduction in diagnosis time did not remain constant but grew as a function of the size of the feature models. For

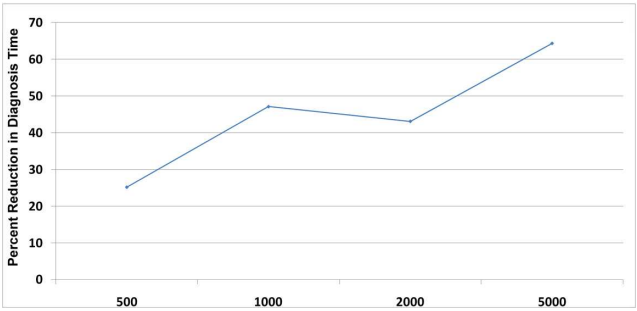

Figure 9: Simplified CSP Formulation Reduction in Diagnosis Time Versus Feature Model Size

example, at 2,000 features, the simplified formulation provided a roughly 43% decrease in average solving time. At 5,000 features, the simplified formulation provided a roughly 64% decrease in average solving time.

The results indicate that for small feature models, the simplified technique can provide a modest improvement in the overall average diagnosis time across a number of feature models. The improvement comes at a reduction in the expressiveness of the goal functions that can be created. For large feature models, however, the simplified CSP formulation provides a significant decrease in average solving time. Moreover, the results indicate that the improvement provided by the simplified CSP formulation is proportional to the size of the feature model, *i.e.*, the simplified CSP formulation saves the most time on the most time-consuming models to diagnose.

6.6. Debugging Scenarios

Staged configuration and viewpoint debugging (Sections 4.3& 4.1) 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 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 ≈ 2.5 seconds and the bounding method by $\approx .1$ seconds.

Cost-based conflict mediation (Section 4.2) 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. Moreover, initiating the diagnosis method with the superset of the configuration participants' desired feature selections also did not impact performance.

7. Related Work

This section compares our work on CURE with related research projects in the areas of automated configuration, autonomic healing, and configuration quality evaluation.

7.1. Constraint Relaxation and Explanation

Junker (Junker, 2004) has investigated the notion of preferred explanations in the context of explaining conflicts in over-constrained CSPs. Junker's work uses a series of user-provided preferences, described as a partial ordering of the constraints, to help compute a set of constraints to relax to make the CSP tractable. CURE is similar to this work in that it uses user-provided information to produce relaxations of the CSP that better fit with the goals of the user. CURE, however, allows developers to specify preferences in terms of the delta, such as the change in cost, between the invalid configuration and the new configuration that is derived. Junker's work does not provide the ability to specify constraint preferences in terms of the difference between an input and output feature model. Junker's research also does not specify how these techniques can be applied to feature model configuration. CURE, however, provides concrete methods for applying constraint relaxation to staged configuration, configuration conflict resolution, and other critical feature modeling topics.

Other constraint relaxation and explanation techniques, such as those developed by Garcia et al. (de la Banda et al., 2003), include methods for finding a minimal set of unsatisfiable constraints. Garcia's concept of minimality searches for the minimal number of constraints. This concept of minimality is valid for some diagnosis cases but not for cost-based diagnostics. This work is somewhat similar to CURE's ability to find the minimal set of changes that can be applied to a configuration to reach a new and valid configuration. Finding a minimal set of unsatisfiable constraints, however, is not specific to feature modeling and does not provide mechanisms for performing cost-based feature diagnostics or other important industrial feature modeling analyses, such as staged configuration debugging. Also, as with with Junker's techniques and constraint relaxation in general, Garcia's diagnosis techniques only propose ways of retracting configuration de-

cisions and not ways of both adding and removing configuration decisions as CURE does. Furthermore, as with other techniques, this approach does not provide the ability to define preferences on the diagnostics that are related to the changes that must be made to the original configuration.

Another series of constraint explanation approaches are based on forms of truth maintenance (Hagg et al., 2006). These approaches record a justification with each constraint that is added to the CSP. When a conflict occurs, users must retract their justifications, which causes all of their associated constraints to be removed, until a satisfiable CSP is reached. CURE allows developers to specify feature selections that can and cannot be changed, which is similar to choosing which justifications to remove. At the same time, CURE allows developers to specify diagnosis preferences that are a function of the delta between the input configuration and output configuration. Truth maintenance systems do not provide this capability, which is crucial for performing optimal cost-based conflict resolution, staged configuration debugging, and software feature evolution analysis.

7.2. Automated Configuration

In prior work, Trinidad et al. (Trinidad et al., 2007) showed how feature models can be transformed into diagnosis CSPs and used to identify *full mandatory features*, *void features*, and *dead feature models*. 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 et al. focus 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.

Debugging techniques for feature models were investigated by Batory et al. in Batory (2005). 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. Moreover, propositional logic-based approaches do not typically provide maximization or minimization of numeric formulas as primitive functions provided by the solver. Since, CURE uses a CSP-based approach, minimization/maximization diagnosis functionality is built-in.

For example, developers may want to determine the cheapest way of correcting a flawed feature selection, where the definition of cost is defined by a complex numerical formula. To perform this type of cost-based correction of invalid feature selections requires the use of numerical maximization/minimization functions. SAT solvers are geared towards boolean logic and not numerical optimization.

Pure::variants (Beuche, 2003), Feature Modeling Plugin (FMP) (Czarnecki et al., 2005a), FeAture Model Analyser (FAMA) (Benavides et al., 2007), and Big Lever Software Gears (Buhrdorf et al., 2003) 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.

Propositional logic has been used to represent software product lines by Mannion Mannion (2002). These techniques are useful for representing product line models, but do nothing to correct flawed feature models. Moreover, they do not take intelligent conflict resolution into consideration. CURE differs in that it can be used

to diagnose and remedy existing, invalid feature selections. The two approaches are complementary, Mannion's techniques can flag flawed configurations and CURE can be used to repair the configuration flaws.

7.3. Autonomic Healing

Dudley et al. (2004) present systems that can autonomically identify and repair system problems. In the event that a system fails, these self-healing solutions are capable of determining the element or elements that are responsible for the failure. Once the nature of a failure is determined, a replacement element is automatically chosen from a set of potential replacement elements, thus allowing the system to recover. Like CURE, this technique is corrective and capable of taking an invalid system configuration, discovering the element or elements that make the configuration invalid, and replacing them with elements autonomically to create a valid system. Unlike CURE, however, this technique does not make use of feature models and is subject to runtime constraints, making it hard to use for SPLs or diagnosis optimization.

7.4. Configuration Quality Evaluation

Other techniques provide mechanisms for reasoning about feature model configurations when the exact non-functional properties of each feature are not known. Zhang et al. (2003) introduce Bayesian Belief Networks, which predict the impact of feature selection on quality attributes by considering the effects of choosing similar variants on the quality of past projects. Jarzabek et al. (2006) present a goal-oriented analysis that places emphasis on the satisfaction of quality attribute during feature selection. These techniques explore a different area of configuration than CURE, the are focused on finding configurations when there is uncertainty about the effects of configuration decisions. In contrast, CURE examines existing configurations, in which the ramifications of selecting a particular feature are already known.

Other modeling approaches, such as COVAMOF (Sinnema et al., 2004) and QRF (Niemelä and Immonen, 2007), examine the ramifications of feature selection on the overall quality of the system. CURE also examines the effect of selecting a feature on the overall quality of the system. Unlike CURE, however, these techniques do not examine how to correct flawed system configurations. CURE is a configuration diagnosis method and not a configuration methodology.

The use of various quality attributes to assess potential system configurations have been investigated. For example, Immonen (Immonen, 2005). introduces a tool for the prediction of availability and reliability for a system configuration. Olumofin et al. (Olumofin and Misic, 2005) provides a method that examines the availability, reliability, performance, and modifiability of potential system configurations. Like CURE, these quality attributes are used to determine which feature to select at each point of variability. Unlike CURE, however, these methods are used in the design phase to construct an initial system configuration, whereas CURE is used to correct flawed feature selections.

Etxeberria and Sagardui (2008) present a manual technique for using domain-relevant quality attributes to evaluate variants in product line architectures. An additional quality feature tree can be added as a branch to extended feature models. These quality features have a predefined qualitative and quantitative impact on the overall system if selected. Feature selection can therefore determine the overall quality of a system configuration and assure that the system possesses necessary functionality. Again, CURE is a corrective technique for restoring validity to flawed feature selections, whereas the method in Etxeberria and Sagardui (2008) is used during the design phase to determine valid initial configurations.

8. Concluding Remarks

Production SPLs rely on large feature models that make it hard to always produce correct configurations. When errors due occur, the lack of configuration debugging tools forces developers to manually pinpoint errors in configurations that can span thousands of features. Moreover, due to the size of many feature models and other fac-

tors (such as supply-chains), SPL configuration may be performed in stages or in parallel (Batory et al., 2006). In these situations, the feature selections of individual developers can conflict and introduce errors in feature models that are even more challenging to identify and correct.

Another challenge of using an SPL is maintaining configurations over time. For example, an SPL that encompasses components obtained through a supply-chain may have features that become unavailable due to vendors exiting the market or discontinuing support. In these scenarios, developers need a way of automating the evolution of existing configurations to new and correct feature selections that take these changes into account.

This paper introduced a technique called *Configuration Understanding and REmedy* (CURE) that transforms invalid configurations into CSPs and automates (1) the diagnosis of invalid configurations and (2) the evolution of existing configurations to meet changing requirement sets. CURE derives the closest valid configuration to an existing configuration that meets a new set of requirements. Moreover, developers can customize CURE to provide optimal cost-based error and evolution diagnostics.

CURE's CSP-based diagnosis model is extensible and can be modified to perform conflict mediation, run faster by using a simplified CSP formulation, or debug from different viewpoints. Our 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 in many production SPLs.

The following are lessons learned from our efforts with CURE thus far:

- **Simplifying the diagnostic CSP can yield substantial improvements in diagnosis time.** A simplified version of CURE's CSP can be used to provide significant improvements in solving speed with a limited reduction in diagnostic flexibility. The reduced flexibility does not allow developers to place limits on only feature selection *or* deselection. If developers do not need to precisely pinpoint one type of change, the simplified diagnostic CSP should be used.

- **CURE can scale to 1,000s of features.** We learned from our experiments with CURE that it can diagnose models with roughly 5,000 features. The results from techniques, such as bounding or CSP simplification, indicate that potential to scale CURE up to larger feature model sizes. The techniques improve CURE's speed by reducing the number of variables that are present in the CSP and placing a precise bound on the search space.

- **It is hard to predict CURE's diagnosis time.** CURE's diagnosis time is dependent on the structure of the constraints in the feature model. For some models, CURE provides very fast diagnosis. In future work, we plan to investigate heuristic diagnostic techniques that can speed CURE's diagnosis time for models it does not perform well on.

- **A good K bound for CURE is hard to ascertain.** 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.

- **Configuration evolution problems can be modeled as configuration diagnosis problems.** Although configuration evolution problems do not necessarily start with an invalid feature selection, we learned that CURE's diagnosis techniques can still be applied successfully to them. When a feature model's constraints are changed or a configuration's feature selection needs to be modified, CURE can be used to automatically modify legacy configurations by model-

ing them as invalid configurations. Automating configuration evolution can provide cost optimal modification planning, which is hard to achieve with manual evolution techniques.

The CURE diagnosis technique has been implemented as part of the ASCENT Design Studio Intelligence project and is available in open-source format from `code.google.com/p/ascent-design-studio` and `www.isa.us.es/fama`.

9. Acknowledgements

We would like to thank Jose Galindo for his hard work implementing CURE in FAMA.

References

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

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

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

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

Beuche, D., 2003. Variant Management with Pure:: variants. Tech. rep., Pure-Systems GmbH, http://www.pure-systems.com.

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

Clements, P., Northrop, L., 2002. Software Product Lines: Practices and Patterns. Addison-Wesley, Boston.

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

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

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

de la Banda, M. J. G., Stuckey, P. J., Wazny, J., 2003. Finding all minimal unsatisfiable subsets. In: PPDP. ACM, pp. 32–43.

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

Etxeberria, L., Sagardui, G., 2008. Variability Driven Quality Evaluation in Software Product Lines. In: Software Product Line Conference, 2008. SPLC'08. 12th International. pp. 243–252.

Hagg, A., Junker, U., O'Sullivan, B., August 2006. A survey of explanation techniques for configurators. In: Proceedings of ECAI-2006 Workshop on Configuration.

Immonen, A., 2005. A method for predicting reliability and availability at the architectural level. Research Issues in Software Product-Lines-Engineering and Management, T. Käkölä and JC Dueñas, Editors.

Jaffar, J., Maher, M., 1994. Constraint Logic Programming: A Survey. Journal of Logic Programming 19 (20), 503–581.

Jarzabek, S., Yang, B., Yoeun, S., 2006. Addressing quality attributes in domain analysis for product lines. In: Software, IEE Proceedings-. Vol. 153. pp. 61–73.

Junker, U., 2004. Quickxplain: Preferred explanations and relaxations for over-constrained problems. AAAI Press / The MIT Press, pp. 167–172.

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

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

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

Niemelä, E., Immonen, A., 2007. Capturing quality requirements of product family architecture. Information and Software Technology 49 (11-12), 1107–1120.

Olumofin, F., Misic, V., 2005. Extending the ATAM Architecture Evaluation to Product Line Architectures. In: IEEE/IFIP Working Conference on Software Architecture, WICSA.

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

Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004. COVAMOF: A Framework for Modeling Variability in Soft-

ware Product Families. LECTURE NOTES IN COMPUTER SCIENCE, 197–213.

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

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

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

White, J., Schmidt, D. C., Benavides, D., Trinidad, P., Ruiz-Cortez, A., Sep. 2008. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In: Proceedings of the Software Product Lines Conference (SPLC). Limerick, Ireland.

Zhang, H., Jarzabek, S., Yang, B., 2003. Quality Prediction and Assessment for Product Lines. LECTURE NOTES IN COMPUTER SCIENCE, 681–695.