Generating Range Fixes for Software Configuration

Yingfei Xiong*, Arnaud Hubaux†, Steven She*, and Krzysztof Czarnecki*
* *Generative Software Development Lab, University of Waterloo, Canada, {yingfei,shshe,kczarnec}@gsd.uwaterloo.ca*
† *PReCISE Research Centre, University of Namur, Belgium, ahu@info.fundp.ac.be*

***Abstract*—To prevent ill-formed configurations, highly configurable software often allows defining constraints over the available options. As these constraints can be complex, fixing a configuration that violates one or more constraints can be challenging. Although several fix-generation approaches exist, their applicability is limited because (1) they typically generate only one fix, failing to cover the solution that the user wants; and (2) they do not fully support non-Boolean constraints, which contain arithmetic, inequality, and string operators.**

This paper proposes a novel concept, *range fix*, for software configuration. A range fix specifies the options to change and the ranges of values for these options. We also design an algorithm that automatically generates range fixes for a violated constraint. We have evaluated our approach with three different strategies for handling constraint interactions, on data from five open source projects. Our evaluation shows that, even with the most complex strategy, our approach generates complete fix lists that are mostly short and concise, in a fraction of a second.

I. Introduction

A growing share of software exposes sophisticated configurability to handle variations in user and target-platform requirements. Large enterprise systems, operating systems, and embedded software need to be tailored to different user needs, hardware, and other requirements. For example, Linux kernel can be configured to run on different hardware systems, and the user typically selects the CPU architecture (e.g., x86 or ARM), the type of filesystem (e.g., ext4 or JFS), and the graphics driver (e.g., ATI or NVIDIA).

These configuration options, also known as features, are usually described with variability modeling languages [1]. For instance, the Linux kernel uses Kconfig, and eCos—an embedded configurable operating system—uses CDL [2]. In academia, a popular language is feature model [3].

Configurators translate these models into interactive configuration interfaces, such as the one in Figure 1, and assist users in arriving at a correct and complete configuration. To do so, configurators detect possible configuration errors and report them. A configuration error is a decision that conflicts with some constraints. Satisfying these constraints can be non-trivial. Variability languages often provide advanced constructs that introduce hidden constraints [2], and constraint rules declared in different places of the variability model may have interactions. The interplay of these factors often leads to very complex situations.

Some configuration tools, like those based on Kconfig, implement an error avoidance mechanism that automatically deactivates an option when a certain constraint is violated. Inactive options are no longer available to the user unless the constraint is satisfied again. Other configurators, like the eCos configurator for CDL (Figure 1), add an interactive resolution mechanism on top of the avoidance mechanism. This approach allows violating some constraints, but proposes a fix for each violated constraint. A fix denotes a set of changes that would restore the consistency of the current configuration.

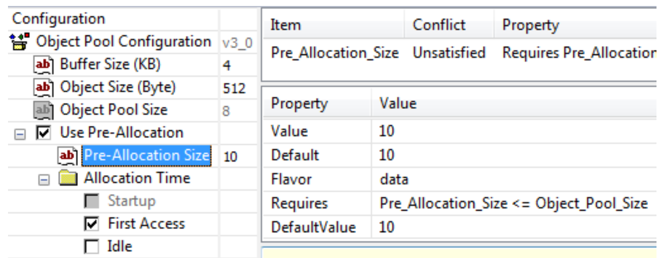

Figure 1. The eCos Configurator

To better understand what challenges are faced by the users of modern configurators, we carried out two surveys, respectively, of Linux and eCos users [4]. Two questionnaires were submitted to forums, mailing lists, and experts with whom we collaborate. In total, we collected answers from 97 Linux users with up to 20 years of experience, and 9 eCos users with up to 7 years of experience. We present here the two challenges that stand out most from this study and that are addressed in this paper:

- **Activating inactive features.** 20% of the Linux users report that, when they need to change an inactive option, they need at least a "few dozen minutes" in average to figure out how to activate it. 56% of the eCos users also consider the activation of an inactive option to be a problem.
- **Fix incompleteness.** Existing configurators generate only one fix for an error. However, there are often multiple solutions to resolving an error, and the user may prefer other solutions. 7 out of 9 eCos users have encountered situations where the *generated fix is not useful*. That claim is corroborated by Berger et al. [2] who report that eCos users complain about the incompleteness of fixes on the eCos mailing list.

Since activating an inactive feature requires satisfying an appropriate constraint, activation is inherently the same as resolving a configuration error. As a result, a possible solution for the above two problems is to generate fixes for both resolving errors and activating features. The set of fixes should be complete so that the user can choose the one he wants.

To achieve this goal, two main challenges need to be addressed. First, our previous study of eCos models [5] shows that non-Boolean operators, such as arithmetic, inequality, and string operators, are quite common in their constraints. In fact, the models contain four to six times more Non-Boolean constraints than Boolean ones. Non-Boolean constraints are challenging since there is often an infinite number of ways to satisfy them. Computing such infinite list of fixes is pointless. Thus, a *compact* and *intentional* representation of fixes is needed. Second, many existing approaches (e.g., [6], [7]) fail to generate a complete list of fixes because they are built upon constraint solvers, which returns only one result per call. To get a complete list of fixes, we need to find a new method to interact with constraint solvers.

This paper proposes a new approach to generating fixes for software configuration. Our contribution is threefold:

- **Range fixes.** We propose a novel concept, *range fix* (Section II), to address the first challenge. Instead of telling users what concrete changes should be made, a range fix tells them what options should be changed and in what range the value of each option can be chosen. A range fix can represent infinite number of concrete fixes and still retains the goal of assisting the user to satisfy constraints. Particularly, we discuss the desired properties of range fixes, which formalize the requirements on the fix generation problem. In addition, we also discuss how constraint interactions should be handled in our framework (Section IV).
- **Fix generation algorithm.** We designed an algorithm that generates range fixes automatically (Section III) to address the second challenge. Our algorithm builds upon Reiter's theory of diagnosis [8], [9] and SMT solvers [10]. Additionally, our algorithm is designed for a general representation of constraints and variables, which makes it potentially useful in other areas.
- **Evaluation with eCos.** Our algorithm is (1) applied on eCos CDL (Section V) and (2) evaluated on 117 constraint violations from five open source projects using eCos (Section VI). The evaluation compares three different fix generation strategies. Even with the most complex propagation strategy, our notion of range fix leads to mostly simple yet complete sets of fixes (83% of the fix lists have sizes smaller than 10, where the size is measured by summing up the number of variables in all the fixes in the list), and our algorithm can also generate fixes for models containing hundreds of options and constraints in an average of 50ms and a maximum of 245ms.

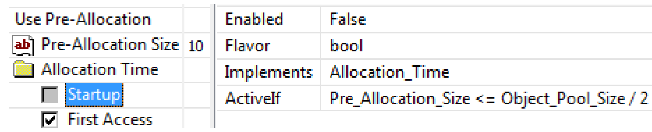

Figure 2. Option "Startup"

We discuss threats to validity in Section VII and the related work in Section VIII. We conclude the paper in Section IX.

II. Range Fixes

Motivating Example. We now motivate our work with a concrete example based on the eCos configurator [11]. Figure 1 shows a small model for configuring an object pool. The left panel shows a set of options that can be changed by the user. The lower-right panel shows the properties of the currently selected option, defined by the eCos model. Particularly, the *flavor* property indicates whether the option is a Boolean option or a data option. A Boolean option can be either selected or unselected; a data option can be assigned an integer or a string value. In Figure 1, "Pre-Allocation Size" is a data option; "Use Pre-Allocation" is a Boolean option.

Besides the flavor, each option may also declare constraints using *requires* property or *active-if* property. When a requires constraint is violated, an error is reported in the upper-right panel. In Figure 1, option "Pre-Allocation Size" declares a requires constraint requiring its value be smaller than or equal to "Object Pool Size", and an error is reported because the constraint is violated.

An active-if constraint implements the error avoidance mechanism. When it is violated, the option is disabled in the GUI and its value is considered as zero. Figure 2 shows the properties of the "Startup" option. This option declares that at most half of the object pool can be pre-allocated. Since this constraint is violated, the "Startup" option is disabled and the user cannot change its value.

Fixing a configuration error or activating an option requires satisfying the corresponding constraints. In order to fix the error on "Pre Allocation Size" in Figure 1, we need to look up the definition of "Object Pool Size". In Figure 3, we see that "Object Pool Size" declares a *calculated* property, meaning that the value of the option is determined by the declared expression and cannot be modified by the user. As a result, the constraint declared on "Pre-Allocation Size" is, in fact, the following:

```
Pre_Allocation_Size <=
  Buffer_Size * 1024 / Object_Size
```

Furthermore, according to the CDL semantics, when an option is inactive, the constraints it declares are not considered by the error checking system. An option is inactive

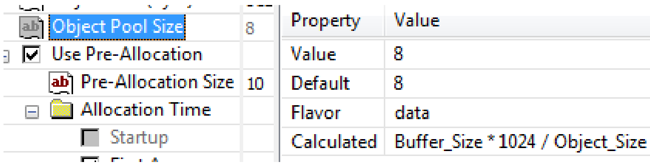

Figure 3. Option "Object Pool Size"

when its active-if constraint is violated or its parent option is deselected. "Pre-Allocation Size" has a parent, yielding the following complete constraint:

```
Use_Pre_Allocation -> (Pre_Allocation_Size <=
  Buffer_Size * 1024 / Object_Size)
```

By analyzing the constraint, we realize that we may fix the error by one of the following changes: decreasing "Pre-Allocation Size", or increasing "Buffer Size", or decreasing "Object Size", or, more simply, disabling the pre-allocation function. Now we could choose one of these possibilities and navigate to the respective option to make the change.

This example shows that there are three sub-tasks for enabling a constraint. First, the user needs to figure out the complete semantic constraint according to the constraint language. Since variability languages often have fairly complex semantics on visibility and value control [2], it is very easy to overlook some part of the constraint. Second, users need to analyze the semantic constraint—such as the complete constraint above—and figure out how to change the options to make it satisfied. In practice, constraints can be very large. One semantic constraint we have found in a CDL model contains 55 option references and 35 constants, connected by 66 logical, arithmetic, and string operators. It is very difficult to analyze such a large constraint. Thirdly, users have to navigate to the corresponding options and make the changes. Real world variability models contain thousands of options, e.g., one reported eCos model [2] contains 1244 options, which makes navigation very cumbersome [4].

Solution. Our approach automatically generates a list of range fixes to help satisfy a constraint. For the error in Figure 1, we will generate the following fixes.

- `[Use_Pre_Allocation := false]`
- `[Pre_Allocation_Size: Pre_Allocation_Size <= 8]`
- `[Buffer_Size: Buffer_Size >= 5]`
- `[Object_Size: Object_Size <= 409.6]`

Each range fix consists of two parts: the option to be changed and a constraint over the options showing the range of values. The first range fix is also a concrete assignment, and will be automatically applied when selected. The other fixes are ranges. If the user selects, for example, the second fix, the configurator will highlight option "Pre-Allocation Size", prompt the range "`<=8`", and ask the user to select a value in the range.

Range fixes automate the three sub-tasks mentioned above. The semantics of CDL constructs is automatically taken into account and the constraint is automatically analyzed. The navigation is also automatically performed when applying a fix. The user only has to choose a fix and decide a value within the range of the fix.

Definitions. Although different variability languages have different constructs and semantics, existing work [2], [12], [13] shows that all variability models can be converted into a set of variables (options) and a set of constraints. Our approach also builds upon this principle.

In essence, a variability language provides a universe of typed variables $\mathbf{V}$ and a constraint language $\Phi(\mathbf{V})$ for writing quantifier-free predicate logic constraints over $\mathbf{V}$. Consequently, a *constraint violation* consists of a tuple (V, e, c), where $V \subseteq \mathbf{V}$ is a set of typed variables; the current configuration e is a function assigning a type-correct value to each variable in V; and $c \in \Phi(V)$ is a constraint over V violated by e. A fix generation problem for a violation (V, e, c) is to find a set of range fixes to help users produce a new configuration e' such that c is satisfied, denoted as $e' \models c$.

Consider the following example of a constraint violation:

$$\begin{array}{lcl} V & : & \{m : \texttt{Bool}, a : \texttt{Int}, b : \texttt{Int}\} \\ e & : & \{m = \texttt{true}, a = 6, b = 5\} \\ c & : & (m \to a > 10) \wedge (\neg m \to b > 10) \wedge (a < b) \end{array} \tag{1}$$

All range fixes we have seen so far change only one variable, but more complex fixes are sometimes inevitable. For example, we cannot solve violation (1) by changing only one variable. Several alternative fixes are possible:

- $[m := \texttt{false},\ \ b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$

The first fix contains two parts separated by ",", each changing a variable. We call each part a *fix unit*. The second fix is more complex. This fix contains only one fix unit, but the range of this fix unit is defined over two variables. When the fix is executed, the user has to choose a value for each variable within the range.

Taking the above forms into consideration, we can define a range fix. A *range fix* r for a violation (V, e, c) is a set of fix units. A fix unit can be either an *assignment unit* or a *range unit*. An assignment unit has the form of "$var := val$" where $var \in V$ is a variable and val is a value conforming to the type of var. A range unit has the form of "$U : cstrt$", where $U \subseteq V$ is a set of variables and $cstrt \in \Phi(U)$ is a satisfiable constraint over U specifying the new ranges of the variables. A technical requirement is that the variables in fix units should be disjoint, otherwise two different values may be assigned to one variable.

We use $r.V$ to denote the set of the variables to be changed in all units. We use $r.c$ to denote the conjunction of the constraints from all units. The constraint from an assignment unit "$var := val$" is $var = val$, and the constraint from a range unit "$U : cstrt$" is $cstrt$. For example, let r be the range fix $[m := \texttt{false},\ b : b > 10]$, then $r.v = \{m, b\}$ and $r.c$ is $m = \texttt{false} \wedge b > 10$.

Applying range fix r of violation (V, e, c) to e will produce a new configuration interactively. We denote all possible configurations that can be produced by applying r to e as $r \triangleright e$, where $r \triangleright e = \{e' \mid e' \models r.c \wedge \forall_{v \in V}(e'(v) \neq e(v) \rightarrow v \in r.V)\}$

Desired Properties. A trivial way to generate a fix from a violation is to produce a range unit where the variables to change are all variables in the constraint and the range of these variables is the constraint itself. For example, the fix for violation (1) could be $[(m, a, b) : (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b)]$. However, such a fix provides no more information than the original constraint. In this subsection, we discuss the desired properties of range fixes.

Suppose r is a range fix for a violation (V, e, c). The first desired property is that a range fix should be correct: all configurations that can be produced from the fix must satisfy the constraint.

Property 1 (Correctness)**.** $\forall e' \in (r \triangleright e),\ e' \models c$

Second, correct range fixes over the same set of variables are often highly overlapping. For example, $[m := \texttt{false},\ b : b > 11]$ is included in $[m := \texttt{false},\ b : b > 10]$, though both are correct. To give users more choice, we would like to present the maximal range of the variables.

Property 2 (Maximality of ranges)**.** *There exists no fix r' for (V, e, c) such that r' is correct, $r'.V = r.V$ and $(r \triangleright e) \subset (r' \triangleright e)$*

Third, even with the above two properties, the number of possible fixes may still be large. Thus, we further rely on a heuristic rule to reduce the number of fixes: a fix should change a minimal set of variables. The reason is that each value currently assigned to a variable is a configuration decision made by the user, and a fix should not unnecessarily break user decisions. For example, $[m := \texttt{false},\ b : b > 10]$ is preferable to $[m := \texttt{false},\ b : b > 10,\ a : a = 9]$ because the latter unnecessarily changes a, which does not contribute to the satisfaction of the constraints.

Property 3 (Minimality of variables)**.** *There exists no fix r' for (V, e, c) such that r' is correct and $r'.V \subset r.V$.*

As a heuristic rule, minimality of variables sometimes excludes possible changes that resolve a violation. For example, given a constraint $i > j$ and a configuration $\{i = 5,\ j = 10\}$, we can find a possible fix $[i := 6, j := 5]$, but this fix is excluded because we can satisfy the constraint by only changing i or j. As a result, it is important to evaluate whether the excluded changes are really needed by users. In our evaluation described in Section VI, the excluded changes are never adopted by the user.

Additionally, after deciding the range over the variables, we would like to represent the range in the simplest way possible. Thus, another possible property is that a fix unit should change as few variables as possible. In other words, no fix unit can be divided into smaller equivalent fix units. We call this property *minimality of (fix) units*.

However, satisfying this property is not easy. Since we treat Φ as a general notion, dividing a range unit into minimal units implies to know the dimension of the solution to an arbitrary system of equations, where no general algorithm is known to exist. As a result, we do not treat this property as a formal requirement that the fix generation algorithm must satisfy. On the other hand, real world constraints are usually not that complex and effective algorithm can be found. As our evaluation will show, although our algorithm cannot absolutely guarantee this property, it effectively satisfies this property on our data set.

Armed with these properties of range fixes, we can define the *completeness* of a list of fixes. Since the same constraint can be represented in different ways, we need to consider the semantic equivalence of fixes. Two fixes r and r' are *semantically equivalent* if $(r \triangleright e) = (r' \triangleright e)$, otherwise they are *semantically different*.

Property 4 (Completeness of fix lists)**.** *Given a constraint violation (V, e, c), a list of fixes L is complete iff*

- *any two fixes in L are semantically different,*
- *each fix in L satisfies Property 1, 2, and 3,*
- *and any fix that satisfies Property 1, 2 and 3 is semantically equivalent to a fix in L*

Thus, a *fix generation problem* is to find a complete list of fixes for a given constraint violation (V, e, c).

III. Fix Generation Algorithm

In Section II we claimed that a fix should change a minimal set of variables and have a maximal range. As a result, our generation algorithm consists of three stages. (i) We find all minimal sets of variables that need to be changed. For example, in violation (1), a minimal set of variables to change is $D = \{m, b\}$. (ii) For each such set of variables, we replace any unchanged variable in c by its current value, obtaining a maximal range of the variables. In the example, we replace a by 6 and get $(m \rightarrow 6 > 10) \wedge (\neg m \rightarrow b > 10) \wedge (6 < b)$. (iii) We simplify the range to get a set of minimal, or close to minimal, fix units. In the example we will get $[m := \texttt{false},\ b : b > 10]$. Stage (ii) is trivial and does not demand further developments. We now concentrate on stages (i) and (iii).

A. From Constraint and Configuration to Variable Sets

To collect all minimal variable sets, we resort to Reiter's theory of diagnosis. Proposed by Reiter [8] and revised by Greiner et al. [9], this theory defines the problem of diagnosis and gives *HS-DAG* algorithm for solving the problem.

Fundamentally, Reiter's theory assumes a constraint set that can be split into *hard* and *soft* constraints. The set of hard constraints are invariable and assumed satisfiable. The set of soft constraints can be altered and also be unsatisfiable. A *diagnosis* is a subset of soft constraints that, when removed from the set, restores the satisfiability of the whole set. The problem of diagnosis is to find all minimal diagnoses from a set of hard and soft constraints.

Given the constraint violation (V, e, c), we convert the problem of finding minimal variable sets to the problem of diagnosis by treating c as a hard constraint and converting e into soft constraints. For example, violation (1) can be converted into the following constraint set.

Hard constraint (c):
[0] $(m \to a > 10) \land (\neg m \to b > 10) \land (a < b)$
Soft constraints (e):
[1] $m = \texttt{true}$
[2] $a = 6$
[3] $b = 5$

To make the whole set satisfiable, we need to remove at least constraints $\{1, 3\}$ or constraints $\{2, 3\}$, which correspond to two variable sets $\{m, b\}$ and $\{a, b\}$.

To find all diagnoses, Reiter's theory uses an ability of most SAT/SMT solvers: finding an unsatisfiable core. An *unsatisfiable core* is a subset of the soft constraints that is still unsatisfiable. For example, two unsatisfiable cores in the above constraint are $\{1, 2\}$ and $\{3\}$.

If we cancel at least one constraint from each unsatisfiable core, we get a diagnosis. The HS-DAG algorithm implements this idea by building a directed acyclic graph (DAG), such that each node is labelled either by an unsatisfiable core or SAT, and each arc is labelled by a constraint that is cancelled. The union of the labels on every path from the root to a SAT node defines a diagnosis.

Figure 4 shows an HS-DAG for the above example. Suppose the constraint solver initially returns the unsatisfiable core $\{1, 2\}$, and a root node is created for this core. Then we build an arc for each constraint in the core. In this case, we build two arcs 1 and 2. The left arc is 1, so we remove constraint [1] from the set, and invoke the constraint solver again. This time the constraint solver returns $\{3\}$. We remove constraint [3] and now the constraint set is satisfiable. We create a node SAT for the edge. Similarly, we repeat the same steps for all other edges until all paths reach SAT. Finally, each path from the root to a leaf is a diagnosis. In this case, we have $\{1, 3\}$ and $\{2, 3\}$.

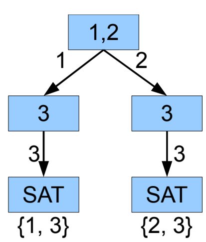

Figure 4. HS-DAG

This process alone cannot ensure that the generated diagnoses are minimal. To ensure it, three additional rules are applied to the algorithm. The details of these rules can be found elsewhere [9], and are omitted here due to space limit. Greiner et al. [9] prove that HS-DAG builds a complete set of minimal diagnosis after applying the three rules.

B. From Variable Sets to Fixes

Equipped with the minimal variable sets, we can replace the variables not in these sets with their configuration values in c (stage(ii)). The purpose of Stage (iii) is to divide this modified constraint into smaller fix units.

Since the operators in the constraints differ from one language to another, this task is essentially domain-specific. Nevertheless, since we assume the constraint language is based on quantifier-free predicate logic, we can perform some general processing. The basic idea is to convert the constraint into conjunctive normal form (CNF), and convert each clause into a fix unit. Yet, we still need to carefully make sure the fix units are disjoint and are as simple as possible.

First, if the constraints contain any operators convertible to propositional operators, we convert them into propositional operators. For example, eCos constraints contain the conditional operator "?:" such as $(m\,?\,a : b) > 10$. We convert it into propositional operators: $(\neg m \lor a > 10) \land (m \lor b > 10)$.

Second, we convert the constraint into CNF. In our example, with $\{m, b\}$, we have $(m \to 6 > 10) \land (\neg m \to b > 10) \land (6 < b)$, which gives three clauses in CNF: $\{\neg m \lor 6 > 10,\ m \lor b > 10,\ 6 < b\}$.

Third, we apply the following rules to simplify the CNF repetitively until we reach a fixed point.

Rule 1. Apply constant folding to all clauses.

Rule 2. If a clause contains only one literal, delete the negation of this literal from all other clauses.

Rule 3. If clause C_1 contains all literals in C_2, delete C_1.

Rule 4. If a clause has the form $v = c$ where v is a variable and c is a constant, replace all occurrences of v with c.

In our example, applying Rule 1 to the above CNF, we get $\{\neg m,\ m \lor b > 10,\ 6 < b\}$. Applying Rule 2 to the above CNF, we get $\{\neg m,\ b > 10,\ 6 < b\}$. No further rule can be applied to this CNF.

Fourth, two clauses are merged into one if they share variables. In the example, we have $\{\neg m,\ b > 10 \land 6 < b\}$.

Fifth, we apply domain specific rules to simplify the constraints in each clause and to divide the clause into smaller, disjoint ones. These rules are designed according to the types of operators used in the constraint language. In our current implementations of CDL expressions, we use two types of rules. First, for clauses containing only linear equations or inequalities with one variable, we solve these equations and inequalities as a system. Second, we eliminate some obviously eliminable operators, such as replacing $a+0$ with a. We also apply Rule 1 and Rule 4 shown above during the process. In the example, the second clause consists of two linear inequalities, we solve the inequalities, getting $\{\neg m, b > 10\}$.

Finally, we convert each clause into a fix unit. If the clause has the form of v, $\neg v$, or $v = c$, we convert it into an assignment unit, otherwise we convert it into a range unit. In the example, we convert $\neg m$ into an assignment unit and $b > 10$ into a range unit and get $[m := \texttt{false},\ b :\ b > 10]$.

Our algorithm ensures the four formal properties. Since in stage (i) HS-DAG gives us all minimal variable sets, Property 3 and 4 hold. Since in stage (ii) we get a range by replacing unchanged variables in the original constraint, the range is ensured to be correct and maximal, and thus Property 1 and 2 hold. Stage (iii) simplifies the range by equivalence transformations, and thus all properties still hold. Our algorithm does not always ensure minimality of units because (1) the CNF simplification rules in the third step of stage (iii) are incomplete (but efficient, as our evaluation will show), and (2) we cannot ensure that the domain-specific rules in the fifth step are complete.

IV. Constraint Interaction

So far we have only considered fixes for one constraint. However, the constraints in variability models are often interrelated; satisfying one constraint might violate another. As a result, we have to consider *multi-constraint* violation rather than single-constraint violation. A multi-constraint violation is a tuple (V, e, c, C), where V and e are unchanged, c is the currently violated constraint, and C is the set of constraints defined in the model and satisfied by e. The following example shows how a fix satisfying c can conflict with other constraints in C that were previously satisfied.

$$\begin{array}{rl} V: & \{m:\texttt{Bool}, n:\texttt{Bool}, x:\texttt{Bool}, y:\texttt{Bool}, z:\texttt{Bool}\} \\ e: & \{m \mapsto \texttt{true}, n \mapsto \texttt{false}, x \mapsto \texttt{false}, \\ & y \mapsto \texttt{false}, z \mapsto \texttt{false}\} \\ c: & m \wedge n \\ C: & \{c_2, c_3\} \text{ where} \\ & \quad c_2 \text{ is } n \rightarrow (x \vee y) \\ & \quad c_3 \text{ is } x \rightarrow z \end{array} \tag{2}$$

If we generate a fix from (V, e, c), we obtain $r = [n := \texttt{true}]$. However, applying this fix will violate c_2.

Note that a multi-constraint violation involves only one violated constraint. If, for example, an eCos configuration contains multiple errors, we treat each of them as a multi-constraint violation and fix them one by one.

Existing work has proposed three different strategies to deal with this problem; each has its own advantages and disadvantages. We now revisit these three strategies, and show that they can all be used with range fix generation by converting a multi-constraint violation into a single-constraint one. In the evaluation section we will give a comparison of the three strategies.

Ignorance. All constraints in C are simply ignored, and only fixes for (V, e, c) are generated. This strategy is used in fix generation approaches considering only one constraint [14]. This strategy does not solve the constraint interaction problem at all. However, it has its merits: first, the fixes are only related to the violated constraint, which makes it easier for the user to comprehend the relation between the fixes and the constraints; second, this strategy does not suffer from the problems of incomplete fix list and large fix list, unlike the two others; third, this strategy requires the least computation effort and is the easiest to implement.

Elimination. When a fix contains changes that violate other satisfied constraints, these changes are excluded from the range of the fix, i.e., any changes with side effect are "eliminated". In the example in violation (2), fix r contains only one change and this change violates c_2. Thus, fix r is eliminated. This strategy is proposed by Egyed et al. [15] and used in their UML fix generation tool.

To apply this strategy to range fix generation, we first find a subset of C that shares variables with c, then replace the variables not in c with their current values in e, and conjoin this subset of constraints with c. For example, to apply the elimination strategy to violation (2), we first find the constraints sharing variables with c, which includes only $c2$, and then replace x and y in c_2 with their current values, getting $c_2' = n \rightarrow \texttt{false} \vee \texttt{false}$. Then we generate fixes for $(V, e, c \wedge c_2')$.

Although the elimination strategy prevents the violation of new constraints, it has two noticeable drawbacks. First, it excludes many potentially useful solutions. Bringing new errors is often inevitable. Simply excluding the changes will only provide less help. In our example, we will get an empty fix set, which does not provide any solution to the current error. Second, since we need to deal with the conjunction of several constraints, the resulting constraint is much more complex than the original one. Our evaluation showed that some conjunctions can contain more than ten constraints. Nevertheless, compared to the propagation strategy, this increase in complexity is still small.

Propagation. When a fix violates other constraints, we modify further variables in the violated constraints to keep these constraints satisfied. In other words, the fix is "propagated" through other constraints. For example, fix r will violate c_2, so we also modify variables x or y to satisfy c_2. Then the modification of x will violate c_3, and we also modify z. In the end, we get two fixes $[n := \texttt{true}, x := \texttt{true}, z := \texttt{true}]$ and $[n := \texttt{true}, y := \texttt{true}]$. This approach is used in the eCos configuration tool [11] and the feature model diagnosis approach proposed by White et al. [7].

To apply this strategy, we first perform a static slicing on C to get a set of constraints directly or indirectly related to c. More concretely, we start from a set D containing only c. If a constraint c' shares any variable with any constraint in D, we add c' to D. We keep adding constraints until we reach a fixed point. Then we conjoin all constraints in D,

Table I
REAL WORLD CONFIGURATION FILES

Architecture	Project	Options	Constraints	Changes
virtex4	ReconOS	933	330	49
xilinx	ReconOS	765	272	53
ea2468	redboot4lpc	658	96	14
aki3068net	Talktic	817	195	3
gps4020	PSAS	535	85	23
arcom-viper	libcyt	771	189	26

Table II
CONSTRAINT VIOLATIONS

Architecture	Erros in defaults	Errors in changes	Activating
virtex4	56	5	15
xilinx	48	1	2
ea2468	8	8	1
aki3068net	26	3	0
gps4020	12	10	4
arcom-viper	26	0	0

and generate fixes for the conjunction. For example, if we want to apply the propagation strategy to violation (2), we start with $D = \{c\}$, then we add c_2 because it shares n with c, next we add c_3 because it shares x with c_2. Now we reach a fixed point. Finally, we generate fixes for $(V, e, c \wedge c_2 \wedge c_3)$.

The propagation strategy ensures that no satisfied constraint is violated and no fix is eliminated. However, there are two new problems. First, the performance cost is the highest among the three strategies. The constraints in real-world models are highly interrelated. In large models, the strategy often led to conjunctions of hundreds of constraints. The complexity of analyzing such a large conjunction is significantly higher than analyzing a single constraint. Second, since many constraints are considered together, this strategy potentially leads to large fixes (i.e., fixes that modify a large set of variables), and large number of fixes, which are not easy to read and apply.

V. IMPLEMENTATION

We have implemented a command-line tool, ECC Fixer[1], generating fixes for eCos CDL using the Microsoft Z3 SMT solver [10]. ECC fixer takes a CDL configuration file as input, and automatically generates fixes for each configuration error. Alternatively, ECC fixer also allows the user to specify an inactive option to activate.

To implement our algorithm, one important step is to convert the constraint in the CDL model into the standard input format of the SMT solver: SMT-LIB [16]. To perform this task, we carefully studied the formal semantics of CDL [13], [17] through reverse engineering from its configurators and documents. However, there are still two problems to deal with. First, CDL is an untyped language, while SMT-LIB is a typed language. To convert CDL, we implement a type inference algorithm to infer the types of the options based on their uses. When a unique type cannot be inferred or type conflicts occur, we manually decide the feature types.

The second problem is dealing with string constraints. The satisfiability problem of string constraints is undecidable in general [18], and general SMT solvers do not support string constraints [10]. Yet, string constraints are heavily used in CDL models. Nevertheless, our previous study on CDL constraints [5] actually shows that the string constraints used in these models employ a set semantics: a string is considered as a set of substrings separated by spaces, and string functions are actually set operations. For example, `is_substr` is actually a set member test. Based on this discovery, we encode each string as a bit vector, where each bit indicates whether a particular substring is presented or not. Since in fix generation we will never need to introduce new substrings, the size of the bit vector is always finite and can be determined by collecting all substrings in the model and the current configuration.

[1] Available at: gsd.uwaterloo.ca/eccfixer

VI. EVALUATION

A. Methodology

To really know whether the approach works in practice, several research questions further need to be answered by empirical evaluation:

- **RQ1**: How complex are the generated fix lists? Does minimality of units hold?
- **RQ2**: Is Property 3 "minimality of variables" a good heuristic rule in practice?
- **RQ3**: How efficient is our algorithm?
- **RQ4**: Does our approach cover more user changes than existing approaches?
- **RQ5**: What are the differences among the three strategies?

The evaluation uses 6 eCos configuration files from 5 eCos-based open-source projects (Table I). Each file targets a different hardware architecture (the first column in Table I); each architecture uses a different mixture of eCos packages, yielding variability models with different options and constraints (columns three and four). The configuration process for a given model starts from the model's default configuration; the last column in Table I specifies the number of changes made to the default configuration in the project.

The evaluation needs a set of real-world constraint violations. Interestingly, the default configuration for each model already contains *errors*—violations of requires constraints. The first column in Table II shows their numbers. The models share common core packages, causing duplicated errors. A set of 68 errors remain after removing duplicates.

To retrieve more violations from user changes, we attempted to recover the sequence of user changes from the revision history of the configuration files. We assume that the user starts from the default configuration and solves errors from defaults by accepting the suggestions from the eCos configurator. We recorded this corrected default configuration as the first version. Then we compared each pair of consecutive revisions to find changes to options. Next we replayed these changes to simulate the real configuration process. Since we do not know the order of changes within a revision, we used three orders: a top-down exploration of the configuration file, a bottom-up one, and a random one. The rationale for the first two orders is that expert users sometimes edit the textual configuration file directly rather than using the graphical configurator. In this case, they will read the options in the order that they appear in the file, or the inverse if they scroll from bottom to top.

We replayed the changes as just explained and collected (i) errors—violating requires constraints—and (ii) *activation violations*. An activation violation occurs when an option value should be changed, but is currently inactive. The last two columns in Table II show the numbers of the resulting violations from changes. After duplicate removal, 27 errors and 22 activation violations remain; together with the first dataset, we have a total of 117 multi-constraint violations.

Finally, we invoked our tool to generate fixes for the 117 violations. For RQ4, we also invoked the built-in fix generator of the eCos configurator on the 27 errors from the user changes. The activation violations were not compared because they are not supported by the eCos configurator. The experiments were executed on computer with Intel Core i5 2.4 GHz CPU and 4 GB memory.

B. Results

We first give the results for RQ1-RQ4 using the propagation strategy. We answer RQ5 by presenting the comparison of the three strategies last.

RQ1: To answer RQ1, we first calculated two basic measures over the 117 violations: the distribution of the number of fixes per violation (see Figure 5) and the distribution of the number of variables changed by each fix (see Figure 6). From these figures we see that most fix lists were short and most fixes changed a small number of variables. More concretely, 95% of the fix lists contain at most five fixes and 75% of the fixes change less than five variables. There was also an activation violation that did not produce any fix. A deeper investigation of this violation revealed that the option is not supported by the current hardware architecture, and cannot be activated without introducing new configuration errors. The extracted changes actually lead to an unsolved configuration error in the subsequent version.

It is still unclear how the combination of fix number and fix size affect the size of a fix list, and how the large fixes and long lists are distributed in the violations. To understand this, we measured the size of a fix list. The size of a fix list is defined as the sum of the number of variables in each fix. The result is shown in Figure 7. From the figure we can see that the propagation strategy did lead to large fix lists. The largest involves 58 variables, which is not easily readable.

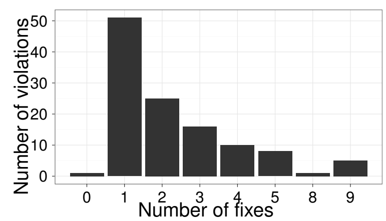

Figure 5. The number of fixes per violation

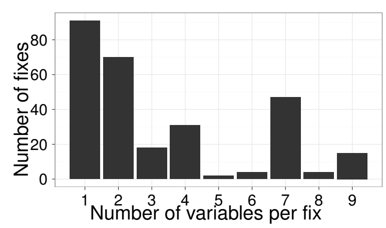

Figure 6. The number of variables per fix

However, the long lists and large fixes tend to appear only on a relatively few number of violations, and the majority of the fix lists are still small: 83% of the violations have fix lists with less than 10 variables.

We also measured the number of variables in each fix unit to understand how large the fix units are. It turned out that every fix unit contains only one variable. This shows that (1) our algorithm effectively satisfies the "minimality of units" property on all the violations, and (2) ranges declared on more than one variable (such as the second fix for violation (1)) never appeared in the evaluation.

RQ2: To answer RQ2, we evaluated how often the final user changes are covered by our fixes. Given an error or activation violation, we examined the change history to identify a subsequent configuration that corrected the problem, and then we checked if the values in the corrected configuration fell within one of the ranges proposed by our generated fixes.

Out of all 49 violation from user changes, a total of 47 had corrections in subsequent revisions. The fixes generated by our tool covered 46 of these violations (98%). An investigation into the remaining violation showed that the erroneous option discussed in RQ1 was responsible for that discrepancy. Since the propagation strategy ensures to

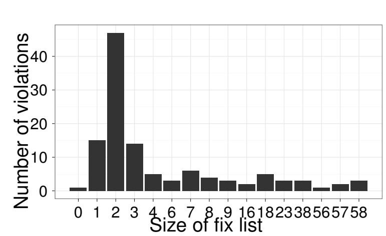

Figure 7. The sizes of fix lists

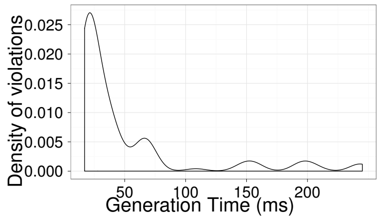

Figure 8. Fix generation time

introduce no new error, the simulated user change was not proposed as a fix.

RQ3: For each of the 117 violations, we invoked the fix generator 100 times, and calculated the average time. The result is presented as a density graph in Figure 8. Most fixes were generated within 100 ms. Some fixes required about 200 ms, which is still acceptable for interactive tools.

RQ4: We measured whether the fixes proposed by the eCos configurator for the 27 errors cover the user changes in the same way as in RQ2. There are 26 out of 27 errors that have subsequent corrections. The eCos configurator was able to handle 19 of the 26 errors, giving a coverage of 73%. Comparatively, our tool covered all 26 errors.

RQ5: As discussed in Section IV, the propagation strategy potentially produces large fix lists. At this stage, we would like to know if the other two strategies actually produce simpler fixes. We compared the size of fix lists generated by the three strategies in Figure 9. The elimination and ignorance strategies completely avoided large fix lists, with the largest fix list containing four variables in total. The elimination strategy changed even fewer variables because some of the larger fixes were eliminated.

We also compared the generation time of the three strategies. For all violations, the average generation time for the propagation strategy was 50ms, while the elimination strategy was 20ms and the ignorance strategy was 17ms. Since the overall generation time is small, it does not make a big difference in tooling.

Next, we want to understand to what extent the elimination strategy affects completeness. In all 117 violations, the elimination strategy generated no fix for 17 violations. This is significantly more than ignorance and propagation, which had zero and one violation with no fix, respectively. We also measured the coverage of user changes using the elimination strategy. In the 47 violations, only 27 were covered, giving a coverage of 57%. This is even lower than the eCos configurator, which generates only one fix, showing that a lot of useful fixes were eliminated by this strategy.

Finally, we want to understand how often the ignorance strategy brings new errors. We compared the fix list of the ignorance strategy with the fix list of the elimination strategy. If a fix does not appear in the list of elimination strategy, it must bring new errors. As a result, 32% of the

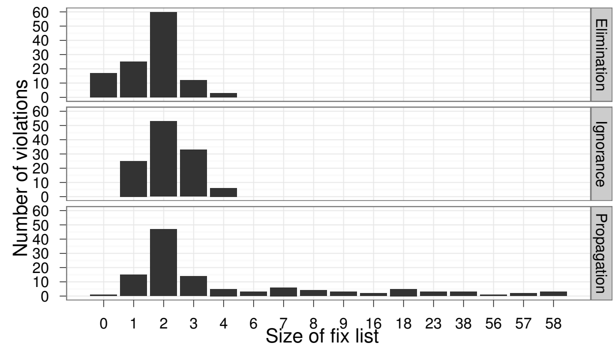

Figure 9. The sizes of fix lists in the three strategies

fixes generated by the ignorance strategy brought new errors, and those fixes were generated from 44% of the constraint violations. This shows that the constraints in practice are usually inter-related and the ignorance strategy potentially causes new errors in many cases.

VII. Threats to Validity

We see two main threats to external validity. First, we evaluated our approach on only one variability language. However, Berger et al. [2] study and compare three variability languages—CDL, Kconfig and feature modeling—and find that CDL has the most complex constructs for declaring constraints, and constraints in CDL models are significantly more complex than those in Kconfig models. Thus, our result is probably generalizable to the other two other languages.

The second threat is that our evaluation is a simulation rather than an actual configuration process. We addressed this threat by using the models of six architectures and configurations gathered from five projects. The configurations and changes have a wide range of characteristics as shown in Tables I and II. However, these changes may not yet be representative of the problems that real users encountered. We hope to address this threat by running a user study in an industry setting in the future.

A threat to internal validity is that our translation of CDL into logic constraints could be incorrect. To address this threat, we have developed a formal specification of CDL semantics in functional style [17], in addition to the one developed by Berger et al. [13]. We have carefully inspected and compared both against each other and tested them on examples with respect to the eCos configurator.

VIII. Related Work

The idea of automatic fix generation is not new. Nentwich et al. [14] propose an approach that generates abstract fixes from first-order logic rules. Their fixes are abstract because they only specify the variables to change and trust the user to chose a correct value. In contrast, our approach also gives the range of values for a variable. Furthermore, their approach only supports "$=$" and "$\neq$" as predicates and, thereby, cannot handle models like eCos. Scheffczyk et al. [19] enhance Nentwich et al.'s approach by generating concrete fixes. However, this approach requires manually writing fix generation procedures for each use of a predicate in every constraint, which is not suitable for variability models, often containing hundreds of constraints. Egyed et al. [15] propose to write such procedures for each type of variable rather than each constraint to reduce the amount of code written and apply this idea to UML fix generation. Yet, in variability models, the number of variables is often larger than the number of constraints. The actual reduction of code is thus not clear. Jose and Majumdar [6] propose an approach to localize faults in imperative programs, and, as an extension to the approach, generate fixes for certain types of faults using heuristic rules. However, their heuristic rules are specific to imperative programs, and it is not clear whether they are applicable to software configuration. Also, they propose at most one fix each time rather than a complete list.

Fix generation approaches for variability models also exist. The eCos configurator [11] has an internal fix generator, producing fixes upon request or on-the-fly when the user makes changes. White et al. [7] propose an approach to generate fixes that resolve all errors in one step. Compared with our work, both related approaches produce one concrete fix rather than a complete list of range fixes. Furthermore, they have very limited support of non-Boolean constraints. White et al.'s approach does not handle non-Boolean constraints at all, while eCos configurator supports only non-Boolean constraints in a simple form: $v \oplus c$ where v is a variable, c is a constant, and $\oplus$ is an equality or inequality operator.

Another set of approaches maintain the consistency of a configuration. Valid domains computation [20], [21] is an approach that propagates decisions automatically. Initially all options are set to an unknown state. When the user assigns a value to an option, it is recorded as a decision, and all other options whose values are determined by this decision are automatically set. In this way, no error can be introduced. Janota et al. [22] propose an approach to complete a partial configuration by automatically setting the unknown options in a safe way. However, both approaches require that the configuration starts with variables in the unknown state. Software configuration in real world is often "reconfiguration" [2], i.e., the user starts with a default configuration, and then makes change to it. In reconfiguration cases, variables have assigned concrete values rather than the unknown state. Furthermore, the related approaches are designed for small finite domains, and it is not clear whether they are scalable to large domains such as integers.

Several approaches have been proposed to test and debug the construction of variability models themselves. Trinidad et al. [23] use Reiter's theory of diagnosis [8] to detect several types of deficiencies in FODA feature models. Wang et al. [24] automatically fix deficiencies based on the priority assigned to constraints. These approaches target the construction of variability models and cannot be easily migrated to configuration.

Others automatically fix errors without user intervention. Demsky and Rinard [25] propose an approach to fix runtime data structure errors according to the constraint on the data structure. Mani et al. [26] use the hidden constraints in a transformation program to fix input model faults. Xiong et al. [27] propose a language to construct an error-fixing program consistently and concisely. Compared to our approach, these approaches also infer fixes from constraints, but they only need to generate one fix that is automatically applied. Completeness is not considered by these approaches.

The HS-DAG algorithm is often used in combination with the QuickXPlain algorithm [28]. The QuickXPlain algorithm computes the preferred explanations and relaxations for over-constrained problems. This combination has been successfully applied in recommender systems to find the most representative relaxations of a set of requirements, i.e., those with highest likelihood of being chosen by the users [29]–[31]. O'Sullivan et al. [32] propose an alternative algorithm for the same problem. The most representative relaxations are then used to propose alternative solutions based on a database of known operational solutions. The filtering of fixes is a possible extension to our work.

IX. Conclusion and Future Work

Range fixes provide alternative solutions to constraint violations in software configuration. They are correct, minimal in the number of variables per fix, maximal in their ranges, and complete. We also evaluated three different strategies for handling the interaction of constraints: ignorance, elimination, and propagation. No single strategy is absolutely better than the others, but on our data set, the propagation strategy provides the most complete fix lists without introducing new errors, and the fix sizes and generation times are within acceptable ranges. However, if more complex situations are encountered, elimination or ignorance can provide simpler fix lists and faster generation time, at the expense of completeness or the guarantee not to introduce new errors.

We highlight two future directions. Regarding evaluation, we still need to evaluate the approach on more languages and models, and with real world user studies. Regarding constraint interaction, since no single strategy works best in all aspects, it would be worth exploring new methods to combine these strategies for better fix lists.

References

[1] L. Chen and M. Ali Babar, "A systematic review of evaluation of variability management approaches in software product lines," *Information and Software Technology*, vol. 53, no. 4, pp. 344–362, 2011.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE*. ACM, 2010, pp. 73–82.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon Univ, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[4] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in Linux and eCos," in *VaMoS*. ACM press, 2012, pp. 149–155.

[5] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski, "A study of non-boolean constraints in variability models of an embedded operating system," in *FOSD*. ACM, 2011, pp. 2:1–2:8.

[6] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *PLDI*. ACM, 2011, pp. 437–446.

[7] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *SPLC*. IEEE Computer Society, 2008, pp. 225–234.

[8] R. Reiter, "A theory of diagnosis from first principles," *Artificial intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[9] R. Greiner, B. Smith, and R. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis," *Artificial Intelligence*, vol. 41, no. 1, pp. 79–88, 1989.

[10] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS/ETAPS*. Springer-Verlag, 2008, pp. 337–340.

[11] B. Veer and J. Dallaway, "The eCos component writer's guide," ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html, 2001.

[12] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC*. IEEE Computer Society, 2007, pp. 23–34.

[13] T. Berger and S. She, "Formal semantics of the CDL language," www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf, 2010.

[14] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE*. IEEE Computer Society, 2003, pp. 455–464.

[15] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *ASE*. IEEE Computer Society, 2008, pp. 99–108.

[16] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," http://www.smtlib.org/, 2010.

[17] Y. Xiong, "Configurator semantics of the CDL language," Generative Software Development Laboratory, University of Waterloo, Tech. Rep. GSDLAB-TR 2011-06-05, 2011, http://gsd.uwaterloo.ca/GSDLAB-TR2011-06-05.

[18] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *TACAS*. Springer-Verlag, 2009, pp. 307–321.

[19] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz, "Managing inconsistent repositories via prioritized repairs," in *DocEng*. ACM, 2004.

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

[21] M. Mendonça, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, University of Waterloo, 2009.

[22] M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva, "How to complete an interactive configuration process?" in *SOFSEM*. Springer-Verlag, 2010, pp. 528–539.

[23] 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*, vol. 81, no. 6, pp. 883–896, 2008.

[24] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei, "A dynamic-priority based approach to fixing inconsistent feature models," in *MODELS*. Springer-Verlag, 2010, pp. 181–195.

[25] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *OOPSLA*. ACM, 2003, pp. 78–95.

[26] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha, "Automated support for repairing input-model faults," in *ASE*. ACM, 2010, pp. 195–204.

[27] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *ESEC/FSE*. ACM, 2009, pp. 315–324.

[28] U. Junker, "QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems," in *AAAI*. AAAI Press, 2004, pp. 167–172.

[29] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, "Consistency-based diagnosis of configuration knowledge bases," *Artificial Intelligence*, vol. 152, pp. 213–234, 2004.

[30] D. Jannach and J. Liegl, "Conflict-directed relaxation of constraints in content-based recommender systems," in *Advances in Applied Artificial Intelligence*. Springer Berlin / Heidelberg, 2006, vol. 4031, pp. 819–829.

[31] A. Felfernig, G. Friedrich, M. Schubert, M. Mandl, M. Mairitsch, and E. Teppan, "Plausible repairs for inconsistent requirements," in *IJCAI*. Morgan Kaufmann Publishers Inc., 2009, pp. 791–796.

[32] B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu, "Representative explanations for over-constrained problems," in *AAAI*. AAAI Press, 2007, pp. 323–328.