A Process for Fault-Driven Repair of Constraints Among Features

Paolo Arcaini
National Institute of Informatics
Japan
arcaini@nii.ac.jp

Angelo Gargantini
University of Bergamo
Italy
angelo.gargantini@unibg.it

Marco Radavelli
University of Bergamo
Italy
marco.radavelli@unibg.it

ABSTRACT

The variability of a Software Product Line is usually both described in the problem space (by using a variability model) and in the solution space (i.e., the system implementation). If the two spaces are not aligned, wrong decisions can be done regarding the system configuration. In this work, we consider the case in which the variability model is not aligned with the solution space, and we propose an approach to automatically repair (possibly) faulty constraints in variability models. The approach takes as input a variability model and a set of combinations of features that trigger conformance faults between the model and the real system, and produces the repaired set of constraints as output. The approach consists of three major phases. First, it generates a test suite and identifies the condition triggering the faults. Then, it modifies the constraints of the variability model according to the type of faults. Lastly, it uses a logic minimization method to simplify the modified constraints. We evaluate the process on variability models of 7 applications of various sizes. An empirical analysis on these models shows that our approach can effectively repair constraints among features in an automated way.

CCS CONCEPTS

• Software and its engineering → Software product lines.

KEYWORDS

automatic repair, fault, variability model, system evolution

ACM Reference Format:

1 INTRODUCTION

Most software systems can be configured in order to improve their capability to address users’ needs. Configuration of such systems is generally performed by setting system parameters [34]. These parameters, or features, can be identified at design time. For instance, in the case of a software product line, the designer identifies the features unique to individual products and features common to all products. Such options can also be decided during compilation time, in order to improve some characteristics of the compiled code (scalability, efficiency, etc.) or to activate/deactivate some functionalities. For example, in the case of preprocessor directives, the programmer can decide which libraries to use, what code to execute and what to ignore etc. Software configurations can also be modified during operation time, when the system is already running. In this case, for example, the parameters can be saved in a configuration file and modified if necessary. Such a configuration file can also be used to decide which features to load at startup.

Constraints exist among system features. They can prohibit system configurations that are dangerous or undesired, or can describe conditions leading to certain properties or errors in code, such as preprocessor errors, parser errors, type errors, and feature effect [30]. Designers, developers, and testers can greatly benefit from modelling features and constraints among them, as it allows to reduce development effort [33] and to identify corner cases of the system under test.

Constraints among features can be modeled using variability models, and imposed on the implementation by means of preprocessor directives, makefiles, etc. These two ways of modeling variability are usually known as problem space and solution space [30]. Fig. 1 presents an example of a variability model containing the constraints among three system features (A, B, and C) that are implemented as preprocessor directives in the C program.

This separation between problem and solution space allows users to model configuration without knowledge about low-level implementation details. On the other hand, these two spaces need to be consistent; code and models, however, are often not kept synchronized, and repairs are needed. In the evolution of product lines, two common types of repair are performed: debugging and program repair, when the variability model is correct but the implementation has to be fixed; and model repair, when the program is correct, but the variability model is outdated. This latter case occurs when
the description of variability is evolved in the implementation, but not in the variability model. This work tackles this problem and proposes a technique to automatically repair variability models. Fig. 2 shows an overview of this context.

In order to detect discrepancies between the problem space and the solution space, classical techniques for testing of propositional formulas (as the constraints of a variability model) can be used: the classical decision and condition coverage, the MCDC [17], fault based criteria [9], and also combinatorial testing [10]. In this paper, we assume that some tests have been generated according to some coverage criterion and some faults (i.e., non-conformances of the model w.r.t. the solution space acting as oracle) have been detected; our aim is to repair the constraints of the variability model in order to remove the faults. We propose an automated process that, upon some failing tests, is able to automatically correct the constraints in such a way that they maintain their original validity for all the configurations, except for those found failing.

The rest of the paper is organized as follows. Sect. 2 presents some basic definitions. Sect. 3 presents the basic repair process and some possible optimizations. Sect. 4 shows the empirical results. Threats to validity are tackled in Sect. 5, whereas an overview of the related work is given in Sect. 6. Sect. 7 concludes the paper and proposes lines for future research.

2 BASIC DEFINITIONS

Definition 1 (Variability Model). A variability model $M$ is made of a set of features $F = \{f_1, \ldots, f_n\}$ and a set of constraints $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ over the features.

The features $F$ represent the system parameters. The expressions in $\Gamma$ identify the features configurations for which the actual system is expected to work.

Definition 2 (Configuration). A configuration (or test) $t$ is a particular assignment of values for all the features $F$. We identify with $t(f_i)$ the value of feature $f_i$ in test $t$. A configuration is valid if it respects the constraints, i.e., $t \models \Gamma$. We also use $\Gamma$ as predicate to check the constraint satisfaction: $\Gamma(t) = \text{true}$ iff $t \models \Gamma$.

Definition 3 (Test suite). A test suite $T$ is a set of tests. We identify with $T_e$ the exhaustive test suite, i.e., the set of all the possible tests.

Definition 4 (Oracle). The oracle function $oracle(t)$ tells whether the configuration $t$ is functionally correct for the system $S$.

We assume that an oracle exists, that tells whether a configuration is valid or not in the real system.

We assume that the set of features $F$ is known and correctly modeled, while the constraints could be faulty.

Definition 5 (Model correctness). We say that the model $M$ is correct if it conforms with the oracle for every possible configuration $t$, i.e., $\forall t \in T_e : \Gamma(t) = oracle(t)$.

Definition 6 (Conformance fault). We say that the model contains a conformance fault if there exists a configuration $t$ such that $\Gamma(t) \neq oracle(t)$.

Definition 7 (Combination). A combination (or partial configuration) $c$ is an assignment to a subset features(c) of all the possible features $F$, i.e., $\text{features}(c) \subseteq \text{features}(f)$. A configuration (or test) is thus a particular combination in which $\text{features}(c) = F$. The value assigned by the combination $c$ to the feature $f$ is denoted as $c(f)$.

Definition 8 (Propositional representation of combinations). A combination $c$ can be expressed in propositional logic by making the conjunction of the truth value assignments of its features:

$$c = \left( \bigwedge_{f \in \text{features}(c)} c(f) \right) \wedge \left( \bigwedge_{f \notin \text{features}(c)} \neg c(f) \right)$$

Definition 9 (Combination containment). A test (or configuration) $t$ contains a combination $c$ if all features values in $c$ are the same in $t$. Formally, $\forall f \in \text{features}(c) : c(f) = t(f)$.

Given a test suite $T$, we identify all the tests containing a combination $c$ as $T(c)$. Formally, $T(c) = \{ t \in T \mid c \subseteq t \}$.

Definition 10 (Combination completeness). Given a test suite $T$ and a combination $c$, we say that $c$ is complete w.r.t. $T$ iff $T(c)$ contains all possible tests containing $c$.

Lemma 2.1. If $c$ is complete w.r.t. a test suite $T$, then it holds $\forall t \in T_e \setminus T(c) : c = false$.

Definition 11 (Failure-containing combination). A combination $c$ is a failure-containing combination (fcc) if:

1. $c$ is contained in at least a failing test of $T$, i.e., $\exists t \in T(c) : \Gamma(t) \neq oracle(t)$.
2. every configuration containing $c$ has the same value in the oracle, i.e., $\forall t \in T(c) : oracle(t) = false \lor \forall t \in T(c) : oracle(t) = true$. In the former case, we call $c$ an under-constraining fcc; in the latter case, an over-constraining fcc.

We further classify a conformance fault as under-constraining fault if it is exposed by an under-constraining fcc, or as over-constraining fault if it is exposed by an over-constraining fcc.
Table 1: Test suites with faults (in gray)

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Test Suite</th>
<th>Revised Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under-constraining fault</td>
<td>A B C M₂ oracle</td>
<td>A B C M₂ oracle</td>
</tr>
<tr>
<td>Over-constraining fault</td>
<td>A B C M₂ oracle</td>
<td>A B C M₂ oracle</td>
</tr>
</tbody>
</table>

Figure 3: Context of the process to repair constraints among features in variability models

In the following, we only consider complete fccs.

Example 1 (Under-constraining fcc). Let’s assume that the oracle is the system (C program) of Fig. 1 with features F = [A, B, C] and that we have generated the test suite shown in Table 1a. Given a faulty model Mᶠ, with only one constraint Γ = {A → B}, we observe only one fault which is represented by the fcc c = A ∧ ¬C. Note that c is a complete fcc that identifies an under-constraining fault, i.e., it proves that the model is under-constrained.

Example 2 (Over-constraining fcc). Consider now a faulty version M₂ of the model in Fig. 1, characterized by Γ = {A → B, C}. Given the test suite shown Table 1b, we detect two over-constraining faults identified by the complete fcc c = ¬A.

3 FAULT-DRIVEN REPAIR

We here propose a process to repair the constraints of a variability model, based on the detection of conformance faults between the model and the system, represented as failure-containing combinations. Fig. 3 shows the context in which our process is applied. We assume that a possibly faulty variability model is translated to a set of boolean formulas representing the constraints. For example, if the model is a feature model, semantic transformations presented in [12] can be used. From a sufficiently large test suite, complete fccs have been identified. To this aim, one can use well-known fault localization techniques like [8, 20]. Our process takes as input the fccs and the constraints and repair them. If the user wants to go back to the initial format of the variability model, she must apply some reverse engineering (which is out of the scope of this work).

We first describe a naïve implementation of the repair process in Sect. 3.1, and we then introduce some optimizations in Sect. 3.2.

3.1 Naïve repair approach

In Def. 11, we distinguish between two types of failure-containing combinations (i.e., under-constraining and over-constraining fcc), depending on how the model fails with respect to the oracle.

We can devise a naïve repair approach that applies a specific type of repair on the base of the fault type:

1. Strengthening repair: in case of under-constraining fcc c, ¬c is added as a new constraint to Γ, i.e., the constraints set Γ’ of the repaired model becomes Γ’ = Γ ∪ {¬c}.

2. Weakening repair: in case of over-constraining fcc c, c is disjuncted with every constraint in Γ, i.e., the constraints set Γ’ of the repaired model becomes Γ’ = Γ ∪ {c}.

Example 3 (Strengthening repair). The under-constraining fault in Ex. 1 is repaired by adding ¬c = (¬A ∧ ¬C) = A → C as a new constraint in Γ. The repaired constraints become Γ’ = {A → B, A → C}.

Example 4 (Weakening repair). The over-constraining fault in Ex. 1 is repaired by adding c = ¬A in disjunction with all the existing constraints, so that the repaired constraints become Γ’ = (A → B) ∨ ¬A, C ∨ ¬A. Note that the first constraint is redundant, as it is equivalent to the original constraint A → B. The only necessary application of the repair is the one in the second constraint, as it correctly allows to have both features A and C assigned to false (as in the oracle).

Theorem 1 (Correctness of the naïve approach). If a combination c is complete w.r.t. its test suite T(c), the repairs applied by the naïve approach to Γ (obtaining the modified constraints set Γ’ are correct, i.e., they remove all existing faults in T(c) and do not introduce new ones, i.e.,

1. ∀t ∈ T(c): Γ’(t) = oracle(t);
2. ∀t ∈ Tᶜ \ T(c): Γ’(t) = Γ(t).

Proof. Let’s consider the two kinds of repairs separately:

- Strengthening repair: the repaired constraints are Γ’ = {γ₁, ..., γᵣ, γᵣ₊₁}, where γᵣ₊₁ = ¬c.
  (1) From the definition of under-constraining fcc, we know that it holds ∀t ∈ T(c): oracle(t) = false. Furthermore, we also know that ∀t ∈ T(c): Γ’(t) = false, because the new constraint γᵣ₊₁ = ¬c falsifies all the tests containing the fcc c. Therefore, it holds ∀t ∈ T(c): Γ’(t) = oracle(t).
  (2) By Lemma 2.1, we know that ∀t ∈ Tᶜ \ T(c): c = false. Therefore, the added constraint γᵣ₊₁ = ¬c is always true in tests Tᶜ \ T(c). Since γᵣ₊₁ has no influence on the evaluation of these tests, it holds ∀t ∈ Tᶜ \ T(c): Γ’(t) = Γ(t).

- Weakening repair: the repaired constraints are Γ’ = {γ₁ ∨ c, ..., γᵣ ∨ c}.
  (1) From the definition of over-constraining fcc, we know that it holds ∀t ∈ T(c): oracle(t) = true. Furthermore, we also know that ∀t ∈ T(c): Γ’(t) = true, because all the constraints γᵣ = γᵣ ∨ c admit all the tests containing the fcc c. Therefore, ∀t ∈ T(c): Γ’(t) = oracle(t).
  (2) By Lemma 2.1, we know that ∀t ∈ Tᶜ \ T(c): c = false. Since c is added as a disjunction to the existing constraints, it leaves the constraints equivalent to the original ones, i.e., ∀t ∈ Tᶜ \ T(c): Γ’(t) = Γ(t).
3.2 Optimized repair approach

The naïve repair approach described in Sect. 3.1 could generate some redundancy, as shown in Ex. 4. Therefore, we introduce techniques for constraint selection and simplification to reduce the potential redundancy generated by the naïve approach. The goal is to make fewer edits as possible to the model, since we assume that a model with fewer edits better preserves domain knowledge.

Therefore, the process performs a pre-processing phase, which selects only the constraints \( \Gamma \) that the constraints to make as few changes as possible to them. Namely, we would like to make as few changes as possible to the model, since we assume that a model with fewer edits better preserves domain knowledge.

Fig. 4 shows the optimized repair approach. It consists of three phases: (1) selection of some constraints to modify, (2) modification of the selected constraints, and (3) simplification of the modified constraints. Note that the process can be iterative if we identified more than one \( fcc \) (as in the experiments in Sect. 4). In the following, we consider one iteration of the process.

3.2.1 Update phase: selection and modification. To preserve the domain knowledge embedded in the constraints, we want the process to make as few changes as possible to them. Namely, we would like that the constraints \( \Gamma \) are updated with the following qualities:

1. possibly no more constraints are added;
2. as many constraints as possible are preserved identical;
3. some constraints can be removed.

Therefore, the process performs a pre-processing phase, which selects only the constraints \( \Gamma_S \subseteq \Gamma \) containing some configurations in common with the \( fcc \ c \), and then modifies them. This phase is specific to the type of repair:

- **Strengthening repair** Only the constraints \( y_i \) sharing at least one feature with the \( fcc \ c \) are selected. Formally, \( \Gamma_S = \{ y_i \in \Gamma \mid \text{features}(y_i) \land \text{features}(c) \neq \emptyset \} \), where \text{features} collects the features contained in a formula. Then, the modification phase updates only one constraint \( y_s \) selected randomly from \( \Gamma_S \), by conjuncting \( \neg c \). The repaired constraints set becomes \( \Gamma' = (\Gamma \setminus \{y_s\}) \cup \{y_s \land \neg c\} \).

- **Weakening repair** Only the constraints \( y_i \) that exclude at least one configuration contained in the \( fcc \ c \) are selected. Formally, \( \Gamma_S = \{ y_i \in \Gamma \mid \text{isSAT}(\neg y_i \land c) = \text{true} \} \), where \text{isSAT} tells whether a formula is satisfiable or not. Then, the modification phase updates all the constraints in \( \Gamma_S \) by disjuncting them with \( c \). The repaired constraints set becomes \( \Gamma' = (\Gamma \setminus \Gamma_S) \cup \bigcup_{y_i \in \Gamma_S} \{ y_i \lor c \} \).

3.2.2 Simplification phase. The constraint simplification procedure aims at reducing redundancy in the repaired model, especially when failure-containing combinations involve many features. A straightforward way to simplify a formula (or make it more readable) is to find the smallest, but equivalent expression. This problem is known as the minimum-equivalent-expression problem [14, 21].

We compared the three existing formula minimization techniques, and one minimization method based on mutations we have implemented (ATGT):

1. JBool\(^1\): a tool that recursively applies logic rules and pre-processing techniques, preserving equivalence [13]: literal removal, negation simplification, and/or reduplication and flattening, child expression simplification, and propagation, De Morgan’s law.
2. Quine-McCluskey (QM) [27], a generalization of the Kar- naugh Maps method. It requires the constraints to be in Disjunctive Normal Form (DNF) and its exponential complexity in the number of features makes it suitable only for small models (up to 15 features).
3. Espresso\(^2\), a faster version of the QM method that relies on some heuristics [37].
4. ATGT\(^3\) [15]: a hill-climbing process we implemented that iteratively mutates a formula randomly and checks it for equivalence.

We propose different methods, as we have seen that, in practice, these techniques often produce different outputs and none of them is guaranteed to generate an output which is always minimal compared to the others.

3.2.3 Correctness. Does the optimized approach produce correct repairs? A repair \( r \) is correct if it is equivalent to the one obtained by the naïve approach.

**Theorem 2 (Correctness of the Optimized Approach).** The optimized approach is correct.

**Proof.** The techniques applied in the simplification phase preserve equivalence. Therefore, we only show that the repairs computed in the update phase are equivalent to those computed by the naïve approach, because the two following properties hold:

---

\(^1\) JBool: https://github.com/bpodgursky/jbool_expressions


• The strengthening repair is correct because of distributivity and commutativity of boolean conjunction: \( y_1 \land \ldots \land y_n \land \ldots \land y_m \land \neg c \equiv y_1 \land \ldots \land (y_n \land \neg c) \land \ldots \land y_m \).

• The weakening repair is by definition equivalent to the naïve approach for all the constraints \( y_j \) that are selected in \( \Gamma_0 \) to be modified in the optimised approach (i.e., it performs the same operation of the naïve approach). It is correct also for each non-selected constraints \( y_j \), since for these constraints it holds \( y_j \lor c = y_j \) (as \( c \) is false in the non-selected constraints): thus, \( y_j \) can be left as is. In fact, by translating this expression to a satisfiability problem, we obtain the condition under which the process does not select the constraint:

\[
((y_j \lor c) \equiv y_j) \iff \neg \text{isSAT}((y_j \lor c) \land \neg y_j) \\
\iff \neg \text{isSAT}((y_j \land \neg y_j) \lor (c \land \neg y_j) \lor (\neg y_j \land \neg c \land y_j)) \\
\iff \neg \text{isSAT}(\neg y_j \land c)
\]

\[\square\]

4 EVALUATION

In order to apply our process, we need a faulty variability model \( M \), a set of failure-containing combinations \( FCC \), and an oracle. For the sake of experiments, we take as oracle another variability model \( M_0 \), instead of the real oracle; in this way, we can also extract the set \( FCC \) by comparing \( M \) and \( M_0 \).

4.1 Benchmarks

We have built two sets of benchmarks: \( \text{BENCH}_\text{MUT} \) with seeded faults, and \( \text{BENCH}_\text{REAL} \) with versioned models.

\( \text{BENCH}_\text{MUT} \) (seeded faults). In order to build this benchmark set, we first selected some models to be used as \( M_0 \), from previous papers and feature model repositories:

• example, from Example 1.

• register, a VSpec model for a register typically found in supermarkets, inspired by [38].

• django, an open source web application framework written in Python. Each Django project has a configuration file loaded at launch time. We considered 12 Boolean parameters (features), with constraints devised in our previous work [18].

• tight_vnc from FeatureIDE repository [28].

In order to obtain the initial faulty model \( M \), we seeded random faults in \( M_0 \) using the following mutation operators:

• RC: removal of a constraint. There are studies showing that this is the most common case in practice [26].

• RL: removal of a literal in a constraint.

• SL: substitution of a literal in a constraint.

We generated 30 faulty versions \( M \) of each model \( M_0 \) (10 with each mutation operator).

\( \text{BENCH}_\text{REAL} \) (versioned models). For this benchmark set, we have considered two versions of variability models of the same system.

We use the second version as oracle \( M_0 \), and the first one as the faulty model \( M \). We picked three models of industrial applications from the SPLot repository\(^4\) [29]:

• the process model rhiscom, between versions 2.0 and 3.0;

• an enterprise resource planner (ERP-SPL);

• a windows accessibility module, between versions 7.0 and 8.0.

Table 2 reports the size of all the faulty models \( M \) to be repaired in the two benchmarks, in terms of number of features, number of constraints, and total number of literals in the constraints. For the constraints and literals of \( \text{BENCH}_\text{MUT} \), it reports the average number across the 30 mutants and the minimum and maximum number between parentheses (the number of features is the same across the mutants).

<table>
<thead>
<tr>
<th>Name</th>
<th># features</th>
<th># constraints</th>
<th># literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>example</td>
<td>3</td>
<td>1.67 (1-2)</td>
<td>3.0 (2-4)</td>
</tr>
<tr>
<td>register</td>
<td>3</td>
<td>1.67 (1-2)</td>
<td>3.87 (2-5)</td>
</tr>
<tr>
<td>django</td>
<td>12</td>
<td>4.6 (4-5)</td>
<td>10.87 (9-12)</td>
</tr>
<tr>
<td>tight_vnc</td>
<td>24</td>
<td>11.67 (11-12)</td>
<td>53.2 (45-55)</td>
</tr>
</tbody>
</table>

Table 2: Benchmarks size

4.2 Failure-containing combinations

For the sake of experiments, we obtain the set \( FCC \) from the faulty model \( M \) (having constraints \( \Gamma \)) and the model we use as oracle \( M_0 \) (having constraints \( \Gamma_0 \)), using the following process:

(1) first, we generate a test showing the difference (i.e., conformance fault) between the two models. The test is built as \( t = \text{getModel}(\Gamma \neq \Gamma_0) \), where \( \text{getModel} \) returns a model of the propositional expression, if it exists, or \( \text{null} \) (in this case, the models are equivalent).

(2) then, we start from \( c \leftarrow t \), and,

• if \( \Gamma_0(t) \) holds, then we do \( c \leftarrow \text{rem}(f, c) \), where \( \text{rem} \) removes the assignment of \( f \) in \( c \) and returns the modified \( c \).

• if \( \neg \Gamma_0(t) \) holds, then we do \( c \leftarrow \text{rem}(f, c) \).

This way we can obtain \( fcs \) that are as minimal as possible, and complete (i.e., the oracle is always true in case of over-constraining \( fcc \), and always false in case of an under-constraining \( fcc \)).

4.3 Repair quality metrics

We want to assess the quality of a repair w.r.t. two goals: (i) simplification of the constraints, and (ii) minimization of the impact of edits. To this aim, we introduce two quality metrics that are used to compare Boolean expressions. We apply them to compare the conjunction of the constraints \( \Gamma \) of the original model \( M \) and the constraints \( \Gamma' \) of the repaired model \( M' \) obtained as output of the approach. The metrics are defined as follows:

\[\text{avg}(\min - \max)\]

---

\(^4\)http://52.32.1.180:8080/SPLot/feature_model_repository.html
We run experiments on the two benchmark sets BENCH\textsubscript{MUT} and BENCH\textsubscript{REAL}: namely, we applied the naïve approach (see Sect. 3.1), the optimized approach (see Sect. 3.2) without the simplification phase (onlySelection), and with the simplification phase (employing the ATGT, Espresso, JBool and QM methods). Experiment code was written in Java and experiments were executed on a Linux PC with Intel(R) i7-3770 CPU (3.4 GHz) and 16 GB of RAM. All reported results are the average of 10 runs with a timeout for a single repair of 1 hour. The code and the benchmarks are available at https://github.com/fmselab/VMConstraintsRepair.

Results of the experiments are reported in Table 3. For benchmarks BENCH\textsubscript{MUT}, results are categorized by the type of mutation. For each benchmark model, the table reports the number and size of strengthening and weakening repairs (note that each repair corresponds to one fcs); moreover, for each process setting, it reports the execution time, and the quality of the final model \( \Gamma' \) in terms of CD and ED distances. Values of the strategies ATGT, JBool and QM for repairing RL and SL mutations of tight_vnc and for all the benchmarks of BENCH\textsubscript{REAL} are not reported, because the experiment exceeded the timeout (T/O) of 1 hour.

We evaluate the process using three research questions.

**RQ1:** *Which quality do the constraints repaired by the process have?*

The main goal of this repair process is to not destroy domain knowledge. We consider the quality measures ED and CD to be proxies for domain knowledge preservation, under the assumption that having fewer edits means more preservation of the domain knowledge contained in the constraints. We therefore consider an approach better than another approach if it has smaller values of the quality measures.

The process (in all its versions) completely repairs all the benchmark models, as the fcs in FCC are complete (see Thms. 1 and 2). However, the quality of the repaired models depends on the adopted repair approach. The optimized approach only using selection (onlySelection) always outperforms the naïve process in terms of quality of the repairs, as it modifies a subset of the constraints, and so the two measures CD and ED for it are always lower. The simplification approaches sometimes allow to obtain better repairs than onlySelection, meaning that they remove some redundancy introduced by the repair; however, there is no simplification method that is always better than the others on all the benchmarks for both measures (except for ATGT that is never worse than Espresso). We observe that, in a few cases, CD and ED are higher for a simplification method w.r.t. onlySelection (e.g., ED of Espresso for register RL): we have checked the example and we found that the simplification has removed some redundancy that was already present in the original model \( M \) so modifying the model more than what done by onlySelection.

**RQ2:** *How efficient is the repair approach?*

Computational time varies significantly for the different approaches and models: from 0.01ms of the smallest models, up to...
245 seconds for the windows model (the biggest model having 335 features and 943 constraints) repaired with the Espresso simplifier.

The naive approach, and the optimized approach without simplification (only selection), have been the fastest approaches; execution times of only selection are higher than the naive approach for big models, as it uses a SAT solver for identifying the constraints that must be repaired (see Sect. 3.2.1). Simplification algorithms are the main responsible for slow performance in terms of computation time. ATGT is the slowest one, as it internally calls a SAT solver several times, and the algorithm is yet in a prototypical stage. The second slowest simplification method is Espresso, but we noticed that it is relatively faster than other methods for large models; indeed, it is the only approach able to simplify all the benchmark models, while the others cannot simplify the biggest mutations obtained for tight,vnc and all the models in BENCHREAL in the given timeout. For small models, JBool is the fastest simplification method, followed by QM; however, they both timeout for large models.

RQ3: Is there a repair type that our process handles more efficiently? We are here interested in investigating whether there is an effect of the type of repair on the performances of the process. In order to better understand the computational cost of the type of repairs, Table 4 reports, for BENCHREAL, the average execution times of strengthening and weakening repairs in the selection and simplification phases, and also the average time of any repair (regardless of the type). We only report the results of Espresso, as it is the only tool that completes before the timeout of 1 hour.

We observe that, in the selection phase, strengthening repairs are faster than weakening repairs: indeed, the former ones only do a syntactical analysis of the constraint, while the latter ones need to call a SAT solver (see Sect. 3.2.1). The average repair time in the selection phase is then influenced by the number of repairs of the two types: in ERP-SPL, since almost all the repairs are weakening (see Table 3), the average time is mostly influenced by them; in windows, instead, most of the repairs are strengthening (see Table 3) and so the average repair time is influenced by them.

Regarding the simplification phase, there is no significant difference between the two types of repairs.

5 THREATS TO VALIDITY
We discuss the threats to the validity of our results along two dimensions.

External Validity. Regarding external validity, a first threat comes from the choice of the variability models on which we performed the experiments. In the benchmarks, we totally selected models of seven applications of different sizes, among them two industrial applications from the SPLIT public repository. Although we have not tested our process on bigger feature models, we believe that the number and variety of input data make the results of our evaluation generalizable to other models of similar size.

In BENCHREAL, we simulated real faults in constraints by enumerating the failure-containing combinations (and thus the single repairs) between two versions of constraints. Such simulated faults may not be accurate with respect to real usages in some scenarios. However, we believe that such results may be generalizable in cases when the faults are automatically detected by testing the updated system implementation, with respect to an outdated model, as we believe that the second version of the model accurately reflects the underlying system implementation.

Internal Validity. Regarding internal validity, a first threat involves the number of experiments and the accuracy of results. To this aim, we executed the experiments 10 times.

Another threat comes from the metrics used to assess the effectiveness and efficiency of our approach, not being a good proxy for domain knowledge preservation. We believe, however, that the chosen metrics well represent the concepts of formula readability and impact of the changes (ED), that may be useful for successive reasoning, for a reverse engineering process from propositional formula to feature model, and for comprehension by the user.

Regarding our approach in general, we have identified the following two threats to validity. The first one regards the applicability of our repair technique. We assume that the variability model is given as a set of constraints, while in general other formats (like feature models) are widely used. However, it is almost always possible to extract the set of features and the constraints among them, so our approach is generally applicable. It is true that it may be not easy to go back from the repaired model to the original format (see Fig. 3), but we try to change the model as little as possible. This should ease the identification of the applied repairs and facilitate the reverse process to extract the final variability model in another format.

The second threat regards the assumption of the fecs completeness. In general, we may find some conformance faults, but it may be not enough, since our process assumes that the fecs are complete. However, we can notice that every failing test is a complete fec regardless of the test suite. Trying to extract a smaller failing combination from a failing test possibly requires new tests, but there already exist several techniques for fault localization that efficiently can do that [20].

6 RELATED WORK
There exist methods to statistically infer constraints from sampled configurations [1, 3, 16, 39–41]: they use a classifier to infer the conditions among parameter values, that determine a particular property, either a parameter above/below a certain threshold (like in [40]), or directly the configuration being accepted or rejected by the system (as in [41]). These machine-learning based methods are well-documented and supported by application studies to real scenarios, such as learning constraints among parameters in SCAD programs that may cause defects of configurable objects to 3D print [3]; and, in the case of RVJX, showing that it is possible to obtain constraints among Boolean or numerical values, to format the paper to meet desired properties, such as a defined page.
limit [2]. Another interesting application of inferring constraints using machine learning is the case of machine temporal and value constraints from rich logs, for event-based monitoring in industrial SoS (Systems of Systems) [24]. The approach, integrated also with techniques from process mining and specification mining, was applied to the automation system of a metallurgical company, and it consists of a ranking phase, based on some validity ratio metrics on the classification tree outcome, in which constraints that are more likely to be accepted by the users appear first.

The constraints learned (or mined) with such machine learning methods achieve a good accuracy (greater than 80% on average [40]), and they are able to completely infer constraints from scratch, or to specialize the model by adding the new inferred constraints to the existing ones. Our approach, however, is focused on performing any kind of repair to an existing set of constraints, and is also able to generalize the model, or apply arbitrary edits (as described in the classification in [42]). Moreover, our proposed method focuses only on the manipulation of existing constraints, and not on the actual detection of the failure-containing combinations, that we assume as input of our process. Unlike our approach, that takes the failure-containing combinations as input, those ML processes also include automatic detection of such fcs (in the form of constraints), given a sample of configurations classified as valid or non-valid [40]. For these reasons, the approaches are not alternative but complementary, as our approach is not comparable to those ML methods; however, as future work, we believe that it could be interesting to combine our process with those machine learning approaches, to have a more complete process for real case scenarios.

A quality-based model refactoring framework assessing the quality of merging operations among SPL models, expressed in UML [36], supports maintainability of models describing relations among features. It represents another approach to model repair, although it is not focused on repairing constraints in propositional logic. There is a comprehensive general work on repair of models by Reder et al. [35], with a method to detect inconsistencies using a validator, and to generate a repair tree representing in a compact way all the different viable actions to repair the model. This approach has been evaluated on UML models and OCL design rules, and is currently integrated in the Model/Analyzer plug-in for the IBM Rational Software Architect (RSA). We believe that our approach, instead, is a particular case of such repair framework in which the repair actions are fixed and determined by the selection and simplification algorithms (i.e., Espresso, QM, etc.), whereas the inconsistency detection is left to the engineer, who has to provide a set of failure-containing combinations in input to our process. However, despite our process has a fixed repair type and in this paper we evaluated its application, it may be possible to integrate the idea of [35] and build a sort of repair-action tree in which the fcs are applied in different order, for example, or with a different simplification method for each fcs, and we believe that the result of following another path in that repair-action tree could give slightly different results (that could be better or could also be worse).

Program repair techniques, such as SemFix [31], GenProg [25], and Par [23] already apply successive patch transformations, but to repair single faults in the code directly.

A process to detect and repair feature models from conformance faults with respect to another model has been presented in [11]. Our work, however, is able to handle arbitrary constraints of a variability model, and adds also the simplification of such modified constraints. The need for a fault-driven constraint repair process was already envisioned in [22], but no experiments were yet performed.

In the classification of edits to variability models presented in [26], our process fits the categories build fix and adherence to changes in code; in the classification of edits to variability models presented in [42], our process is able to address all kind of edits: in the case of arbitrary edits, it achieves them by applying specialization (what we call strengthening repair) and generalization (what we call weakening repair) sequentially.

A different technique for feature model repair in the context of system evolution, with different versions of systems, used mutation operators to make the model meet a specific update request [6, 7]; however, that approach does not handle arbitrary constraints, and does not guarantee to completely fulfill the update request, and thus repair the model. We believe that that approach could be extended with our process, to be able to repair not only the feature tree, but the constraints as well.

Repair of constraints has usage also in other contexts, such as the repair of parameter values of timed automata clock guards, by applying tests and specializing the constraints [5]; and in the detection of constraints among parameters that let the built attack string trigger an XSS vulnerability in the system [19].

7 CONCLUSION

We proposed a process that, given a (faulty or outdated) variability model, and the faults in terms of failure-containing combinations, identifies the constraints involved in the fault, repairs them according to the oracle value, and simplifies them to make the edit minimal. We conducted an empirical evaluation on 7 models of different sizes, and found that the process of selecting only some constraints is indeed more effective than the naïve approach that modifies all of them. Moreover, we observed that simplification approaches can further improve the quality of the repair. However, their applicability is limited by the model size, as most of them do not scale on big models.

As future work, we plan to adapt our approach to larger models and to include in the evaluation the performances of reverse engineering the final constraints into a variability model (in case the repairs affected the structure of the initial variability model). As future work, we also want to address the current limitations; for example, by designing better selection and simplification strategies, by extending the method to non-boolean variables, and by including new simplification techniques. Furthermore, in order to better preserve domain knowledge, we plan to design an approach that interacts with domain engineers, for instance by highlighting implicit constraints as in [4].

ACKNOWLEDGMENTS

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

REFERENCES
