Automated Repairing of Variability Models

Paolo Arcaini
Charles University
Faculty of Mathematics and Physics,
Czech Republic
arcaini@d3s.mff.cuni.cz

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

Paolo Vavassori
DIGIP
University of Bergamo, Italy
paolo.vavassori@unibg.it

ABSTRACT

Variability models are a common means for describing the commonalities and differences in Software Product Lines (SPL); configurations of the SPL that respect the constraints imposed by the variability model define the *problem space*. The same variability is usually also captured in the final implementation through implementation constraints, defined in terms of preprocessor directives, build files, build-time errors, etc. Configurations satisfying the implementation constraints and producing correct (compilable) programs define the *solution space*. Since sometimes the variability model is defined after the implementation exists, it could wrongly assess the validity of some system configurations, i.e., it could consider acceptable some configurations (not belonging to the solution space) that do not permit to obtain a correct program. We here propose an approach that automatically repairs variability models such that the configurations they consider valid are also part of the solution space. Experiments show that some existing variability models are indeed faulty and can be repaired by our approach.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; • **General and reference** → *Validation*;

ACM Reference Format:
Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2017. Automated Repairing of Variability Models. In *Proceedings of SPLC '17*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3106195.3106206

1 INTRODUCTION

Software Product Lines (SPLs) are families of products that share some common characteristics, and differ on some others [9]. Software product line engineering consists in the development and maintenance of SPLs by taking into account their commonalities and variability. *Software variability*, i.e., the possibility to customize/configure the software for different needs, is particularly important, as it permits to adopt the software in different contexts, so making it more *reusable* [22].

The variability of SPLs is usually already described at design time by using *variability models* [26], whose usefulness is nowadays widely recognized [25]. They are used to represent SPLs by specifying the *constraints* existing between the different *features* of the system, so defining the valid combinations of features; for example, a feature can require or exclude the presence of another feature, or constraints can be expressed over groups of features. The set of configurations that are allowed by a variability model defines the *problem space* [11]. Different kinds of variability models have been proposed, both in academia and in industry, as KConfig, CDL [10], and feature models [7]. SPLs users derive concrete products from variability models using configurators that support the intelligent propagation of choices and auto-completion. Practitioners have also developed a series of automated tools which supply some functionalities of model validation and analysis [12]. These tools ease the management of large configurable systems that can easily have thousands of features with complex constraints.

The variability described in the problem space must be also captured by the system implementation in order to make it customizable/configurable. Different solutions (*implementation constraints*) are employed to express variability in code: preprocessor directives, build files, build-time errors, etc. The set of system configurations respecting all the implementation constraints is called *solution space* [11]: system configurations of the solution space permit to obtain correct compiled programs that can be executed. Differently from the problem space that is compactly described by one design artefact (i.e., the variability model), the solution space definition is scattered among different artefacts. However, it is also possible to build a model for the solution space by "mining" the implementation constraints from build-time errors, build files, and preprocessor directives (e.g., #IFDEF); this is what has been done in [20] using the `FARCE` tool which is based on `TypeChef` [15]. Therefore, in the following we assume to have a model describing the problem space (the variability model) and a model describing the solution space.

Most of the times, variability models are written when the implementation already exists and the solution space already defined; in these cases, the variability model is written in order to make the customization of an existing system easier. In practice, the problem space could not conform to the solution space on some configurations: the solution space could allow to concretize configurations that are not allowed in the problem space, and the problem space could include some configurations that do not produce correct (i.e., compilable) programs. While we still consider the former case acceptable (the variability model imposes further constraints w.r.t. the implementation because of design decisions), the latter is not acceptable, because the variability model would allow us to build systems that are not feasible: in this case, we consider the variability model *faulty*. We here aim at automatically repairing faulty variability models. We present an approach in which a (logic representation of a) variability model is repeatedly modified by the application

SPLC '17, September 25-29, 2017, Sevilla, Spain
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5221-5/17/09…$15.00
https://doi.org/10.1145/3106195.3106206

of some *repair*s. Each repair is a syntactical modification of the variability model such that the new model allows a proper subset of the configurations of the original model, and it no longer allows some configurations that were wrongly accepted in the original model; however, the new model still allows all the configurations that were correctly evaluated by the original model. The process stops when the variability model only accepts configurations that are also accepted by the solution space, or if it is no more possible to modify the model.

We found that some variability models used in practice are indeed faulty (as also found in [14]) and can be repaired by our technique.

The paper is organized as follows. Sect. 2 provides some background on variability models, introduces definitions regarding the structure of the variability models considered in this work, and gives our definitions of configurability fault and of correctness for variability models. Sect. 3 presents the notion of repair of a variability model, and Sect. 4 describes the approach we propose (based on repairs) to remove faults from variability models. Sect. 5 reports the experiments we have done to evaluate the approach, and Sect. 6 discusses possible threats to the validity of the approach. Sect. 7 reports some related work, and Sect. 8 concludes the paper.

2 BASIC CONCEPTS AND DEFINITIONS

The problem space is defined in terms of a *variability model* that identifies the features of a system and their relations. There exist different kinds of variability models [26], like KConfig and CDL [10], and more academic variability models like feature models [7]. All these notations and languages provide a representation in terms of propositional logic, where each feature is a propositional variable and each relation is mapped to a particular propositional formula. For example, the translation from feature models to propositional formulas is presented in [26]. In this work, we do not restrict ourselves to one particular kind of variability model, but we consider all of them taking advantage of their common representation in propositional logic.

The solution space, instead, is identified by the constraints imposed by the system itself. For example, it could be extracted from the source code by analysing the preprocessor directives as proposed by TypeChef [15]. We assume that also the solution space has a representation in terms of propositional logic.

In the following, we identify with P the model of the problem space (the variability model of a given system), and with S the model of the solution space. In this work, we focus on repairing a possibly faulty variability model P. $F = \{a, b, c, \ldots\}$ is the set of features used in the variability model or in the implementation (or in both). From now on, we see F as a set of propositional variables, and P and S as logical formulas defined over F. As we will see in Sect. 2, P is a conjunction of formulas having a particular form. For S, instead, we do not assume any particular form.

Definition 2.1 (Configuration). A *configuration* is a truth assignment to each propositional variable in F.

If in a configuration the truth value of a feature f is *true*, it means that the feature is selected, otherwise that it is not selected. Therefore, a *configuration* can be seen as a subset of the features F. We use $\mathcal{P}$ and $\mathcal{S}$ to identify the problem space and the solution space, i.e., the set of models of P and S.

Definition 2.2 (Product). A *product* is a configuration c accepted by P, i.e., $c \in \mathcal{P}$.

The mapping of any variability model in propositional logic produces a set of *constraints* that, for our purposes, we divide in two families: *core* constraints are those that can be generated by variability models of any kind, while *notation-specific* constraints are generated only by some kinds of variability model. Such partition allows us to be as comprehensive as possible, since we propose an approach that works on core constraints but we do not exclude variability models containing other peculiar constraints. We define $P = \bigwedge_{i=1}^{n} C_i \wedge \Theta$, where C_i are core constraints and Θ notation-specific constraints. Each core constraint C_i in P can only be of this schema (where a, b, and b_i are features):

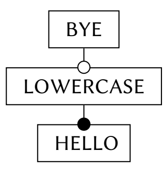
- *Constant feature*: a. The feature must be selected in all the products.
- *Disabled feature*: $\neg a$. The feature cannot be selected in any product.
- *Requires*: $a \Rightarrow b$. Whenever a is selected, also b must be selected.
- *Excludes*: $a \Rightarrow \neg b$. If a is selected, b cannot be selected.
- *OR*: $a \Rightarrow \bigvee_{i=1}^{n} b_i$. If a (called *father*) is selected, at least one b_i (called *children*) is selected.

Core constraints are sufficient to express all the core relationships (like feature relationships, hierarchies, and grouping) in most variability modeling notations. In particular, group relationships that restrict the number of selectable sibling features if their parent is selected can be mapped to our core constraints. As outlined in [10], classical group relationships are XOR when exactly one child is selected, OR for at least one, MUTEX for at most one. XOR and OR groups can be mapped to an OR constraint plus a number of suitable *excludes* and *requires*, while the MUTEX group is modeled only using *excludes* and *requires*. For instance, an OR group of a feature model is represented by a constraint of shape $a \Rightarrow \bigvee_{i=1}^{n} b_i$ and by a sequence of constraints $b_i \Rightarrow a$ requiring that a is selected whenever one of its children is selected.

The shape of constraints in Θ depends on the considered variability model notation. For extended feature models, for example, constraints modeling the *cardinality* relations are notation-specific and could be mapped to Θ (following, for example, the approach suggested in [27]). Notation-specific constraints, although part of P, will not be modified by our approach. Note that the benchmarks we used for our experiments (see Sect. 5.1), representative of medium-size and big variability models, do not have notation-specific constraints. Recent studies found that core constraints are able to represent variability models of real software products [10].

Example 2.3. Fig. 1 reports an example of P (on the left), given as feature model, and S (on the right), given as C code with preprocessor directives. Feature models are a well-accepted means for expressing requirements at an abstract level. They are applied to describe variable and common properties of products in a product line, and to derive and validate configurations of software systems [25]. In the feature model P shown in the figure, the root BYE is always selected, LOWERCASE is an *optional* feature (it *can* be selected only if BYE is selected), and HELLO is a mandatory feature (it *must* be selected whenever LOWERCASE is selected). The only constraint in S is that LOWERCASE can be selected only if HELLO is selected,

Feature model P



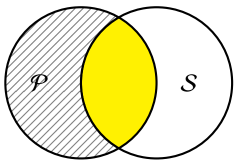
Source code S

```
#ifdef HELLO
char* msg = "Hello!\n";
#endif
#ifdef LOWERCASE
msg[0] = 'h';
#endif
#ifdef BYE
char* bye = "Bye\n";
#endif
```

As Boolean formulas

BYE LOWERCASE $\Rightarrow$ BYE HELLO $\Rightarrow$ LOWERCASE LOWERCASE $\Rightarrow$ HELLO	LOWERCASE $\Rightarrow$ HELLO

Figure 1: Example of P and S



🟨 $\mathcal{P} \cap \mathcal{S}$: correct products
▨ $\mathcal{P} \cap \overline{\mathcal{S}}$: faults

Figure 2: Correct products and faults

otherwise there will be a compilation error. We assume that, in this example, P contains more constraints than S due to some design decisions.

Although P and S do not need to be equal (in fact they very often differ), the designer wants that every configuration allowed by P, i.e., every product, is permitted by S as well. In terms of sets $\mathcal{P}$ and $\mathcal{S}$, their relation is given in Fig. 2. The intersection between $\mathcal{P}$ and $\mathcal{S}$ represents products that are correctly instantiated in S (called *correct products*), while configurations that are in $\mathcal{P}$ but not in $\mathcal{S}$ represent products that cannot be actually realized by the real system, i.e., they represent faults in P, since P wrongly considers them as products.

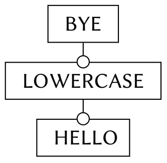
Definition 2.4 (Variability model correctness). P correctly configures S if and only if $\mathcal{P} \subseteq \mathcal{S}$.

Example 2.5. In Fig. 1, the model P correctly configures S. In every product of P, if LOWERCASE is true, then HELLO is true. There are configurations that would be allowed by S but are not permitted by P. For instance, LOWERCASE and HELLO true, but BYE false. This is *not* a fault. With this configuration the code will compile, but still it is not allowed by P for a number of reasons (the code may do something wrong, perform unsafe operations, or for design reasons).

Definition 2.6 (Configurability fault). P contains a *configurability fault* if and only if $\mathcal{P} \cap \bar{\mathcal{S}}$ is not empty.

Example 2.7. If the user had designed P as shown in Fig. 3 (i.e., by setting HELLO optional), then the model would be faulty. If we consider the configuration {BYE, LOWERCASE}, it is accepted by P but not by S (compilation error).

Feature model P



As Boolean formula

BYE
LOWERCASE $\Rightarrow$ BYE
HELLO $\Rightarrow$ LOWERCASE

Figure 3: A faulty P

▧ $\overline{\mathcal{P}} \cap \mathcal{P}'$: products added by P' (to be empty for NA)
🟨 $\mathcal{P} \cap \overline{\mathcal{P}'}$: products removed by P' (to be not empty for RE)
▨ $\mathcal{P} \cap \mathcal{S} \cap \overline{\mathcal{P}'}$: products of P, confirmed by S, removed by P' (to be empty for PR)

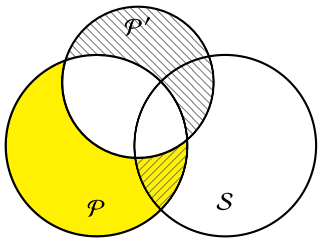


Figure 4: Relation between $\mathcal{P}$, $\mathcal{P}'$, and $\mathcal{S}$

Knowing whether a variability model is correct or not may be not enough, since one can be interested in finding how much a model is faulty. For this reason, we introduce a measure of model bugginess by defining an index measuring the conformance of P w.r.t. S.

Definition 2.8 (Conformity index). We define *conformity index* as

$$CI(P, S) = \frac{|\mathcal{P} \cap \mathcal{S}|}{|\mathcal{P}|}$$

It measures how many products of P can be correctly instantiated by S. We aim at having conformity index equal to 1; in this case, $\mathcal{P}$ would be a subset of $\mathcal{S}$ and so no configuration rejected by S would be accepted by P. Note that computing the conformity index could be difficult in practice, as it requires to compute the set of products of P and S. However, we aim at having a technique that guarantees to improve the index (regardless the possibility to compute it).

3 VARIABILITY MODELS REPAIRS

We here tackle the problem of detecting and removing configurability faults from a variability model. We want to achieve this by applying some modifications, called *repairs*, to P.

Definition 3.1 (Repair). A *repair* is a syntactic modification of P that permits to obtain a different syntactically correct P'.

3.1 Variability model repairs properties

When we apply a repair to a variability model P in order to obtain the repaired P', we modify the products of P as shown in Fig. 4 in terms of the sets $\mathcal{P}$, $\mathcal{P}'$, and $\mathcal{S}$. We want that P' actually repairs P, i.e., it removes some faults of P without introducing any new undesired product. We formalize repair requirements by the following properties:

NA *Not adding*: P' cannot allow configurations that were not already allowed by P. In terms of sets of Fig. 4, the set $\overline{\mathcal{P}} \cap \mathcal{P}'$ must be empty, i.e., $\mathcal{P}' \subseteq \mathcal{P}$.

RE *Removing*: P' must reject some configurations originally allowed by P. In terms of sets of Fig. 4, the set $\mathcal{P} \cap \overline{\mathcal{P}'}$ must be not empty, i.e., $\mathcal{P} \not\subseteq \mathcal{P}'$.

PR *Preserving*: P' must not refuse correct products allowed by P and by S. In terms of sets of Fig. 4, the set $\mathcal{P} \cap \mathcal{S} \cap \overline{\mathcal{P}'}$ must be empty, i.e., $\mathcal{P} \cap \mathcal{S} \subseteq \mathcal{P}'$.

WF *Well-formed*: P' must still represent a variability model, i.e., it can only add constraints that can be mapped back to constructs of the original variability model.

We call *correct repair* a transformation of P in P' that guarantees all the previous properties, i.e., a well-formed variability model such that $\mathcal{P}'$ is a proper subset of $\mathcal{P}$ and still contains all the correct configurations of $\mathcal{P} \cap \mathcal{S}$.

Remark. A naïve approach to produce repairs would be to encode each configuration c which is wrongly accepted by P (i.e., it holds $c \models P \land \neg S$) as a formula $asFormula(c)$[1] and add it to P as $\neg asFormula(c)$ to obtain P'. In this way, P' would exclude exactly all the products of P that cannot be instantiated by S. However, $\neg asFormula(c)$ may violate property WF (i.e., it may be not expressible using the core constraints defined in Sect. 2). Moreover, the number of such wrong configurations could be huge.

3.2 Proposed repairs

We here propose some possible repairs to P in order to obtain a *better* (in terms of conformity index) variability model P'. We only propose repairs for the core constraints of P; in this way, we are proposing an approach for repairing any variability model[2]. The proposed repairs are (where a, b, and b_i are features):

`addCF` *Make a feature constant*: Adding a constraint a to P. Feature a is required in all products $\mathcal{P}'$.

`addDF` *Disable a feature*: Adding constraint $\neg a$ to P. Feature a is disabled in all products $\mathcal{P}'$.

`addREQ` A *requires* constraint $a \Rightarrow b$ is added to P.

`addEXC` An *excludes* constraint $a \Rightarrow \neg b$ is added to P.

`addOR` An OR constraint $a \Rightarrow \bigvee_{i=1}^{n} b_i$ is added to P. This repair is applicable only if P already contains the *requires* constraints $b_1 \Rightarrow a, \ldots, b_n \Rightarrow a$ and no other OR constraint in which a is the antecedent.

`rmORchild` In an OR constraint of P, a child b_j is removed: $a \Rightarrow \bigvee_{i=1}^{n} b_i$ becomes $a \Rightarrow (\bigvee_{i=1}^{j-1} b_i \lor \bigvee_{i=j+1}^{n} b_i)$. All the requires constraints $b_i \Rightarrow a$ (also modeling the corresponding OR block of variability models) are not modified, so the removed child b_j still requires a.

Note that all previous repairs guarantee NA since they can only restrict $\mathcal{P}$, i.e., they can remove some products but not add new ones. Indeed, all repairs that add a new constraint to P do not add products to $\mathcal{P}$ by the semantics of the logical conjunction. Note that also removing a child from an OR does not add products because $\models (a \Rightarrow \bigvee_{i \in \{1,\ldots,n\} \setminus \{j\}} b_i) \Rightarrow (a \Rightarrow \bigvee_{i \in \{1,\ldots,n\}} b_i)$.

Moreover, all previous repairs guarantee WF, i.e., applying a repair to a model P obtained from a variability model of a given type (i.e., feature model, or KConfig model, etc.) produces a model P' that can be traced back to a variability model of the same kind. For feature models, for example, `addCF`, `addDF`, `addREQ`, `addEXC`, and `addOR` have a straightforward mapping using a corresponding construct in the feature model notation: for instance, `addCF` can be obtained by adding a *requires* constraint $r \Rightarrow a$ between the root r and the constant feature a (or, if a is not already present in the feature model, adding a as mandatory child of r). Mapping `rmORchild` requires to remove the child in the corresponding OR block in the feature model; we obtain an *unstructured* feature model that, however, can be aptly restructured using an abstract feature or similar techniques [2, 13]. Similar arguments can be carried on for each variability model notation we consider in this paper.

Therefore, in order for a repair to be correct, we only need to check properties RE and PR. We can express these two properties by means of logic operators (set inclusion becomes logical implication and set intersection becomes logical conjunction):

RE $\not\models P \Rightarrow P'$
PR $\models P \land S \Rightarrow P'$

In the following, our approach will be based on the manipulation of logic formulas.

Example 3.2. Applying the `addREQ` repair LOWERCASE $\Rightarrow$ HELLO to the faulty P shown in Fig. 3 produces the variability model in Fig. 1 that does not contain any fault.

3.3 Repair attributes

As seen in the previous section, for a repair to be correct we have to check properties RE and PR. Checking these two properties has two useful characteristics.

Single use. If a repair is not applicable to P (i.e., it is not correct because it violates RE or PR), it will be also not applicable to any P' obtained from P by the application of correct repairs. This allows us to try to apply each repair only once.

Fast check. Each repair can be seen as a constraint δ that is added to P[3], i.e., $P' = P \land \delta$. Therefore, in order to prove $\models P \land S \Rightarrow P \land \delta$, we just need to prove $\models P \land S \Rightarrow \delta$. This can be efficiently done in an SMT solver as explained in Sect. 4.1.

3.4 Repair precedence

Applying the repairs in different orders may lead to different final results. We now want to define the order among repairs that allows to obtain the *best* repaired model in terms of conformity index (see Def. 2.8).

[1] *asFormula* is a function that, given a model m, returns the conjunction of the variables having value *true* in m and the negation of the variables having value *false* in m.
[2] Note that if the fault is contained in the notation-specific constraints Θ, our approach could be not able to completely repair P.

[3] Note that this is true also for repair `rmORchild` that modifies existing constraints. For each repair of this kind, we can provide an equivalent repair that adds a constraint instead of modifying an existing one.

$$\begin{tikzcd}
P \arrow[rrr, dashed, "\mathtt{apply}(P,R_B)"] \arrow[d, dashed, "\mathtt{apply}(P,R_A)"] & & & P_B \arrow[d, dashed, "\mathtt{apply}(P_B,R_A)"] \\
P_A \arrow[r, dashed, "\mathtt{apply}(P_A,R_B)"'] & P_{AB} \arrow[rr, phantom, "\Longrightarrow"] & & P_{BA}
\end{tikzcd}$$

Figure 5: Repair precedence – $R_A \preceq R_B$

We claim that a family of repair operators R_A must be applied before a family R_B if applying R_A and then R_B removes more products than applying R_B and then R_A. In this case, we say that R_A has precedence w.r.t. R_B (i.e., $R_A \preceq R_B$).

Definition 3.3 (Repair precedence). R_A has *precedence* w.r.t. R_B (formally $R_A \preceq R_B$) if it holds

$$\mathtt{apply}(\mathtt{apply}(P, R_A), R_B) \Rightarrow \mathtt{apply}(\mathtt{apply}(P, R_B), R_A)$$

where $\mathtt{apply}(P, R)$ is a function that applies constraints R to P (those that respect RE and PR) and returns the modified model P'. The situation is depicted in Fig. 5.

We can formally prove the following precedences:

- $R_{\mathtt{addCF}} \preceq R$ for any repair family R
- $R_{\mathtt{addDF}} \preceq R$ for any repair family R
- $R_{\mathtt{addREQ}} \preceq R$ for any repair family R
- $R_{\mathtt{addEXC}} \preceq R$ for any repair family R, except for $R_{\mathtt{rmORchild}}$
- $R_{\mathtt{addOR}} \preceq R_{\mathtt{rmORchild}}$

Making a feature constant, disabling a feature, and adding a requires constraint (i.e., addCF, addDF, and addREQ) do not require the presence of any other constraint, and the presence of the constraints they introduce do not avoid the application of other repairs; therefore, the corresponding families can be applied at any time during the repair process. However, applying one of these repair families after any other repair family does not permit to find a *better* (i.e., more constrained) model; actually, it could avoid the creation of other repairs: for example, creating repairs $R_{\mathtt{addREQ}}$ after repairs $R_{\mathtt{addOR}}$ could avoid the creation of some OR constraint, since there could not be all the implications needed to represent the corresponding OR block of variability models.

Also $R_{\mathtt{addEXC}}$ can be applied before any other repair family, except for $R_{\mathtt{rmORchild}}$. For this couple of repair families, we cannot provide any precedence: indeed, there are cases in which it is more convenient to apply $R_{\mathtt{addEXC}}$ first, and others in which it is the other way round.

$R_{\mathtt{addOR}}$ has precedence w.r.t. $R_{\mathtt{rmORchild}}$ since removing children of ORs is more effective if there are more ORs to modify.

Moreover, all the constraints of a given family R can be applied in any order since for each $r_1, r_2 \in R$, it holds $r_1 \preceq r_2 \wedge r_2 \preceq r_1$. Indeed, there is no family of repairs that produces constraints that can be exploited for building repairs of the same family.

3.5 Repair witness

When we (partially) repair P in P', we can also generate a *witness* w, i.e., a configuration that holds in P and does not hold in S and in P', i.e., $w \models P \wedge \neg S \wedge \neg P'$. Such witness can be used to demonstrate the fault in the original model, for example when doing a bug report (see RQ6 in Sect. 5).

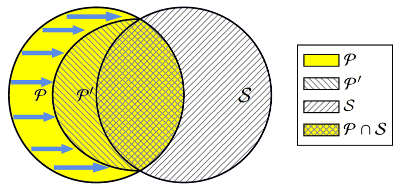


Figure 6: Proposed approach

Algorithm 1 Iterative repairing algorithm

```
Require: $P$: problem space constraints
Require: $S$: solution space constraints
$\phi_{cont} \leftarrow P \wedge \neg S$
$\phi_{RE} \leftarrow P$
$\phi_{PR} \leftarrow P \wedge S$
while $\phi_{cont} \neq false$ do
    $r \leftarrow$ getNextRepair($P$)                ▷ Get next repair
    if $r = null$ then
        return $P$                  ▷ There are no more possible repairs
    end if
    $\delta \leftarrow$ getConstraint($r$)
    if $\phi_{RE} \Rightarrow \delta$ then
        continue                         ▷ No product is removed
    end if
    if $\phi_{PR} \Rightarrow \delta$ then        ▷ Are correct products preserved?
        $\phi_{cont} \leftarrow \phi_{cont} \wedge \delta$
        $\phi_{RE} \leftarrow \phi_{RE} \wedge \delta$
        $\phi_{PR} \leftarrow \phi_{PR} \wedge \delta$
        $P \leftarrow$ apply($P$, $r$)                    ▷ Apply repair $r$
    end if
end while
return $P$
```

4 ITERATIVE REPAIRING PROCESS

We here present an approach that tries to repair a faulty variability model. The proposed approach is informally depicted in Fig. 6. The idea is that we want to repeatedly modify model P into a model P' till P' coincides with $P \wedge S$.

The approach is formally presented in Alg. 1. The approach first builds three logic formulas:

- ϕ_{cont} for checking if the model is not completely repaired, i.e., if there are still products in $\mathcal{P} \cap \overline{\mathcal{S}}$;
- ϕ_{RE} for checking property RE, i.e., that some product allowed by P is removed in the repaired model;
- ϕ_{PR} for checking property PR, i.e., that the products allowed by $P \wedge S$ are not removed in the repaired model.

Then, the following instructions are repeatedly executed till the model is completely repaired, i.e., until ϕ_{cont} becomes unsatisfiable:

- a new repair r is generated (the order in which repairs are generated respects the repair precedences identified in Sect. 3.4). A repair consists in a constraint δ and the information regarding how to apply it (either adding it to the existing constraints or substituting it for an existing one). If no new repair is available, the process terminates returning P

Table 1: Translation to an incremental SMT solver

	Pseudocode	SMT implementation
1	Boolean formula ϕ	logical context ctx_ϕ
2	$\phi \neq false$	$\texttt{sat}(ctx_\phi)$
3	$\phi \leftarrow \phi \wedge \delta$	$\texttt{assert}(ctx_\phi, \delta)$

4	$\phi \Rightarrow \delta$	**function** isImplied(ctx_ϕ, δ)
		    push(ctx_ϕ)
		    assert(ctx_ϕ, $\neg\delta$)
		    isImplied $\leftarrow \neg$sat(ctx)
		    pop(ctx_ϕ)
		    return *isImplied*
		end function

as non-completely repaired model; otherwise, the constraint δ of the repair r is considered.
- if δ is already implied by P (i.e., property RE is violated), the repair r is not considered and the next iteration of the loop is taken;
- if δ is an acceptable constraint since it does not remove correct products (i.e., property PR is not violated as δ is implied by $P \wedge S$), it is conjuncted with each of the three formulas and the corresponding repair r is applied to P (i.e., δ is applied to P according to the information contained in r). Otherwise, r is not considered and the next iteration of the loop is taken.

After the while loop, the algorithm returns the repaired P.

Note that Alg. 1 always terminates, either because the loop condition becomes false or there are no more possible repairs. Since the number of models of ϕ_{cont} decreases at every iteration and the number of possible repairs is finite, the algorithm is guaranteed to terminate either because ϕ_{cont} becomes unsatisfiable or all the possible repairs have been considered.

Note that the algorithm does not guarantee to produce a complete correct variability model. It could terminate before finding the correct model because there are no more applicable repairs. This is expected because it is well known that the logic of variability models (a subset of propositional logic) may be not complete w.r.t. propositional logic [26] that is used instead by S.

4.1 Approach implementation in SMT solver

The approach described in Alg. 1 has been implemented in Java, using an SMT solver (namely Yices) for logical reasoning. The mapping from the logical notation used in the pseudocode to SMT statements is shown in Table 1:

(1) The three formulas ϕ_{cont}, ϕ_{RE}, and ϕ_{PR} are modeled by means of three contexts ctx_{cont}, ctx_{RE}, and ctx_{PR}.

(2) Checking whether ϕ is not false is done by checking whether the corresponding context ctx_ϕ is `satisfiable`.

(3) Conjuncting a new constraint δ to an existing formula ϕ (lines 14, 15, and 16) is done using the command `assert` of δ in the context ctx_ϕ.

(4) Checking whether δ is implied by ϕ_{RE} and ϕ_{PR} (lines 10 and 13) is done by asserting $\neg\delta$ in the corresponding context: if the context becomes unsatisfiable, δ is implied. The context is saved and restored, before and after the checking, by means of commands `push` and `pop`.

Table 2: Benchmarks

Benchmark	# P features	# P constraints	# S features	# Total features ($\lvert F\rvert$)
`uClibc`	114	100	138	165
`eCos`	514	736	322	649
`BusyBox`	610	596	783	801
`Linux`	5736	10452	4389	5913

5 EXPERIMENTS

5.1 Benchmarks

We took the benchmarks (variability models and solution space models) considered in [20]:
- `uClibc`[4]: it is a C library for developing Linux embedded systems. It has a KConfig model as variability model.
- `eCos`[5]: it is a free open source real-time operating system intended for embedded applications. It has a CDL model as variability model.
- `BusyBox`[6]: it combines tiny versions of many common UNIX utilities into a single small executable. It has a KConfig model as variability model.
- `Linux`: it is a general-purpose operating system. It has a KConfig model as variability model.

In [20], variability models are provided using their logical representation and since they use core constraints, they are completely captured by our notation. Solution space models, instead, have been *mined* from build-time errors, the effects of features in build files, and the structure of the code (e.g., #IFDEF usage); since they are also expressed in propositional logic, they can be treated by our approach. Table 2 shows, for each benchmark, the number of features and constraints of the variability model, the number of features of the solution space model, and the total number of features.

All the experiments have been executed on a Linux PC with two Intel(R) Xeon(R) CPU E5-2630 (2.30GHz) and 64 GB of RAM.

5.2 Research questions

We here answer some research questions in order to evaluate our approach.

RQ1 How many repairs are applied?

Table 3 reports, for each benchmark and for each repair family, the number of applied repairs.

As the table shows, the number of applied repairs is between 25 and 278, which makes the manual inspection of the repairs feasible. Note that a single fault in a variability model (e.g., the usage of a wrong block construct) can produce a very high number of wrong configurations (as shown in [14]); so trying to discover faults in configurations by, for example, manually analysing them is

[4]https://www.uclibc.org/
[5]http://ecos.sourceware.org/
[6]https://www.busybox.net/

Table 3: Applied repairs

Benchmark	addCF	addDF	addREQ	addEXC	addOR	rmORchild	Total
uClibc	4	13	6	34	2	2	61
eCos	0	88	185	5	0	0	278
BusyBox	0	1	17	12	1	0	31
Linux	25	N/A	N/A	N/A	N/A	N/A	25

unfeasible. Instead, we are able to apply few repairs also when the number of wrong configurations is very high. Note that the applied repairs could introduce some anomalies [8], as dead features or redundant constraints, that could make the model less readable; however, the repaired model can be polished at the end using some technique that removes such anomalies [8].

RQ2 How many repairs are generated and how many of them are not applied because they are not correct?

Table 4 reports, for each benchmark and for each repair family, the number of generated repairs and those discarded because violating PR or RE properties. The table shows that PR and RE properties efficiently filter the generated repairs: only a few of them are actually applied (those that are correct repairs), while the others are discarded. Note that more repairs are discarded by PR than by RE, although PR is checked after RE in the approach in Alg. 1. This means that generally repairs are able to remove some products (i.e., RE is not violated), but very often they also remove correct products (i.e., PR is violated).

Note that, for the Linux variability model, we obtained the correct model during the application of addCF repairs; therefore, all the other repairs have not been generated.

RQ3 How long does the repair process take and does it actually repair the model?

Table 5 shows, for each benchmark, the time taken by the repair process and the final result (whether the variability model has been partially or completely repaired). Our technique repairs all the four models. In three cases, the repair is not complete: either our repairs are not powerful enough to remove faults in the models or the core constraints are not expressive enough to represent S. For the Linux variability model, instead, we are able to achieve conformity index equal to 1 (i.e., to completely repair the model). Probably the Linux variability model has fewer faults than the others since it is widely used. However, we were still able to find some configurability faults.

As expected, the execution time is proportional to the number of features of the initial variability model. However, the process terminates as soon as it finds a correct model: therefore, although the Linux benchmark is the biggest one, it does not have the highest repair time since it is completely repaired only using addCF repairs. In the worst case, our algorithm terminates in 1h30min.

RQ4 How is the variability model modified?

Table 6 shows the initial and the final number of constraints for all the benchmarks. As expected, the number of constraints always increases because most of the repairs add a new constraint and no repair removes any constraint. However, since the number

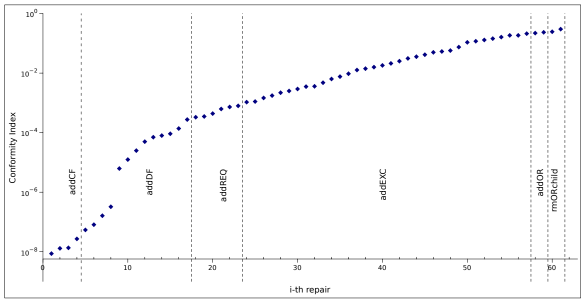


Figure 7: uClibc – **Conformity index**

of repairs is small, also the number of final constraints remains manageable. Moreover, our repairing technique is able to introduce constraints of all types, including OR constraints that are the base for some group relationships of variability models [10].

RQ5 How does the conformity index grow with the repairs application?

We were able to measure the conformity index only for the uClibc benchmark. Fig. 7 shows how the conformity index of the uClibc benchmark grows after each repair application. The plot also reports when (i.e., in which iterations) a repair family is applied. It is apparent that the repair family that is successfully applied for more times than the others is addEXC.

Table 7 reports the same data of Fig. 7: it shows the *conformity index* (from 0 to 1) obtained after the application of each repair family (in the order they were applied), and the *family gain factor* of conformity, i.e., the increment w.r.t. the value of the conformity index before the application of the repair family. The repair families that contribute more during the process are addDF and addEXC. This, however, may depend on the order in which the repairs families are applied. For this reason, we also computed the conformity index that can be obtained by applying only *one* repair family in the process: Table 8 shows, for each repair family, the obtained final conformity index and the gain factor w.r.t. the initial conformity index when we applied only repairs of one family. The addEXC family permits to obtain the best results; indeed, as seen in Sect. 3.4, addEXC can be applied since the beginning of the process, as it is not strictly preceded by any other repair family (except for rmORchild w.r.t. whom we cannot provide any precedence). However, also other families, as addCF, addDF, addREQ (those not relying on existing constraints of P), have this property; they do not have such good performances as addEXC because they are either too strong (and get filtered by PR) or too weak (and get filtered by RE). The rmORchild repairs have no effect when applied alone, since they are effective only if applied after the addOR repairs (at least in our examples).

RQ6 How difficult is to apply a repair back to the original variability model?

We are interested in investigating how a repair produced by our technique can be applied back to the original variability model. We consider the variability model specified for the uClibc library when the target architecture is x86_64 (file Config.x86_64). We have taken the first repair produced for the model, which is an addCF for the

Table 4: Generated repairs (G) and those discarded because of PR and RE

Benchmark	addCF			addDF			addREQ			addEXC			addOR			rmORchild			Total		
	G	PR	RE	G	PR	RE	G	PR	RE	G	PR	RE	G	PR	RE	G	PR	RE	G	PR	RE
uClibc	165	160	1	165	143	9	24251	22674	1571	11465	10024	1407	16	13	1	9	3	4	36151	33085	3005
eCos	649	649	0	649	514	47	362861	329496	33180	157063	131674	25384	105	103	2	0	0	0	521980	463003	58699
BusyBox	801	801	0	801	800	0	639422	639240	165	319583	319510	61	75	74	0	0	0	0	961261	961004	226
Linux	26	0	1		N/A			N/A			N/A			N/A			N/A		26	0	1

Table 5: Execution time and final result of the repair process

Benchmark	Time (sec)	Final result
uClibc	31	Partially repaired
eCos	690	Partially repaired
BusyBox	5241	Partially repaired
Linux	1060	Completely repaired

Table 6: Number of constraints (I: Initial, F: Final)

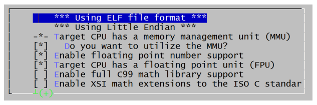
Benchmark		Constant feature	Disabled feature	Requires	Excludes	OR	Total
uClibc	I	0	0	80	15	0	100
	F	4	13	86	49	2	159
eCos	I	0	0	660	21	0	736
	F	0	88	845	26	0	1014
BusyBox	I	0	0	579	17	0	596
	F	0	1	596	29	1	627
Linux	I	0	0	9455	144	0	10452
	F	25	0	9455	144	0	10477

Table 7: Conformity index growth – uClibc

order	repair	conformity index	family gain factor
	initial	5.70×10^{-9}	
1	addCF	2.70×10^{-8}	4.7
2	addDF	2.75×10^{-4}	1.02×10^{4}
3	addREQ	7.94×10^{-4}	2.8
4	addEXC	0.21	266
5	addOR	0.23	1.1
6	rmORchild	0.30	1.2

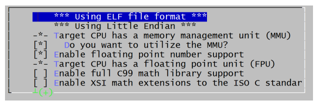
Table 8: Conformity index of single repair families – uClibc

	final conformity index	conformity gain factor
addCF	2.7×10^{-8}	4.74
addDF	5.7×10^{-5}	1.00×10^{4}
addREQ	7.94×10^{-4}	1.39×10^{5}
addEXC	2.73×10^{-2}	4.78×10^{6}
addOR	7.73×10^{-9}	1.36
rmORchild	5.7×10^{-9}	1



```
    *** Using ELF file format ***
    *** Using Little Endian ***
-*- Target CPU has a memory management unit (MMU)
[*]   Do you want to utilize the MMU?
[*] Enable floating point number support
[*] Target CPU has a floating point unit (FPU)
[ ] Enable full C99 math library support
[ ] Enable XSI math extensions to the ISO C standar
(+)
```

(a) Original menuconfig



```
    *** Using ELF file format ***
    *** Using Little Endian ***
-*- Target CPU has a memory management unit (MMU)
[*]   Do you want to utilize the MMU?
[*] Enable floating point number support
-*- Target CPU has a floating point unit (FPU)
[ ] Enable full C99 math library support
[ ] Enable XSI math extensions to the ISO C standar
(+)
```

(b) Repaired menuconfig

Figure 8: menuconfig for uClibc

feature UCLIBC_HAS_FPU (i.e., UCLIBC_HAS_FPU must be always selected). Firstly, as double check, we have generated a witness *w* (see Sect. 3.5) of the repair, and we have checked that *w* identifies an actual wrong product: we have compiled the library without that option selected and we indeed found a compilation error. This confirms that the repair has removed at least one configuration that was not implementable, but that was allowed by the original model[7]. Then, we have modified the original variability model file in order to find the modification necessary to have this feature mandatory. By applying the proposed patch to the faulty menuconfig (see Fig. 8 (a)), we were able to produce a correct menuconfig where the user cannot unselect the feature UCLIBC_HAS_FPU, as shown in Fig. 8 (b). Finally, we have filed a bug report at uClibc bugzilla website[8] (see Sect. 3.5); at the time of writing, we have not received any reply.

6 THREATS TO VALIDITY

The solution space model is extracted from the implementation using a research prototype software (FARCE), that could not capture all the constraints [21], i.e., the retrieved solution space model could accept configurations that cause build-time errors; indeed, FARCE is used to analyse artefacts written in decades-old C-Dialects [21], but does not cover all corner cases (e.g., some GNU C extensions, some unusual build-system patterns). However, all the constraints identified by FARCE are correct, i.e., FARCE is sound (it rejects only faulty configurations, i.e., those causing build-time errors) but not complete (it could also accept faulty configurations). The incompleteness of FARCE does not affect the applicability of our approach, but only the quantitative results: indeed, all the configurations we remove from the problem space (by applying repairs) are correctly rejected also by the solution space model (since FARCE is sound). Moreover, FARCE is not the only approach able to find constraints among features in variability models. For instance, in [28], a machine learning approach infers product-line constraints from an oracle that is able to assess whether a given product is correct; we could use this approach to have more precise solution space models.

[7]Note that the witness is not required for repairing the variability model, but it is useful to provide a proof of the configurability fault.
[8]https://bugs.busybox.net/show_bug.cgi?id=9011

In RQ5, we only considered the growth of the conformity index for `uClibc`; indeed, for computing the index we have used BDDs to count the products of the variability model, and this approach did not scale to the (much bigger) other benchmarks. Therefore, we cannot draw any definitive conclusion on the strength of the different repair families to improve the conformity index; however, we can still prove that all of them improve it.

Applying the repair back to the original variability model could be difficult, as it could be done in multiple ways (at least one way is guaranteed to exist by the fact that each repair guarantees WF) and the user should decide which one to apply in order to preserve the readability of the model. However, in RQ6, we have shown that repairing an existing faulty variability model is feasible. As future work, we plan to devise a technique that assists the user in fixing the original variability model.

7 RELATED WORK

This work is inspired by our previous work [5] in which we tried to automatically detect and remove configurability faults in feature models using an approach based on mutation [4]. Although the aims of the two works are similar (i.e., removing configurability faults), the two approaches are quite different. Firstly, in [5] we target only feature models, while here we try to be as much comprehensive as possible by considering the types of constraints shared by the different variability model notations. Secondly, the previous work does not guarantee to improve the conformity index of the model since it could worsen it in particular cases, while the current approach can only improve it. Thirdly, in [5] mutation families (similar to the repairs of this work) are all continuously applied to the model at each step of the fault removal process. This causes the generation of a lot of model modifications, while in this work we apply each repair at most once. Finally, the current work considers real industrial case studies, while our previous work only considered more academic benchmarks.

Another approach trying to identify inconsistencies between the problem space and the solution space is proposed in [16] in which the authors check whether the implementation constraints correctly capture the constraints of an existing feature model: they derive a feature model from the implementation and check whether this is conformant with the original feature model. The approach differs from ours in several aspects. They only consider feature models, while we consider any variability model kind; they suppose that, in case of inconsistency, the fault is in the implementation, while for us is the other way round; finally, they need to generate a new model and compute the structural difference w.r.t. the initial one to show the inconsistency, while we only need to apply few repairs directly to the initial model to remove configurability faults.

A different approach trying to automatically repair feature models is presented in [14]. The approach starts from a feature model and, through a continuous cycle of test-and-fix, improves it in order to reduce the number of its wrong constraints; the approach uses configurations derived both from the model and from the real system and checks whether these are correctly evaluated by the feature model. The main difference with our approach is that we do not rely on the real implementation, but on the model representing the solution space: therefore, we can avoid considering the single configurations (whose number could be huge), but we can manipulate only constraints.

Another way to repair variability models is explored by Temple et al. [28]. They use machine learning techniques to infer the constraints of a product line. They randomly generate products from the product line and apply machine learning to identify which combinations of features are likely to produce faulty products.

There are some works that do not attempt to repair variability models but, nonetheless, try to isolate or at least study faults in them. For instance, in [1], 42 bugs of the Linux kernel (derived from bug reports) due to variability are analysed; each bug is related to a configuration that does permit to compile the system correctly, but such that the obtained program produces a runtime error. They do not provide any automatic way to fix them.

Other works on variability models are not interested in finding faults w.r.t. the implementation, but in analysing variability models per se, for example for studying their evolution. In [17], the authors study the evolution over time of the Linux variability model both from a quantitative and from a qualitative point of view, identifying how the number of features and constraints grew over time, and how the structure of the model was modified by designers. In [23], the authors found that, during the evolution of the Linux kernel between releases 2.6.32 and 2.6.33, 35% of the features removed from the variability model continued to exist elsewhere in the code: our approach could be applied to keep the variability and the system implementation consistent in case of evolution.

Other approaches are interested in mining constraints for synthesizing either a variability model or a solution space model. Nadi et al. [20] focused their research on deriving constraints from variability models, and from implementation constraints (code, build files, preprocessor directives, etc.); we have used the constraints they derived as benchmarks for our approach. Their main interests are to find how many implementation constraints can be automatically extracted, to detect how these constraints relate to the constraints of the variability models, and to classify the constraints of the variability model. Differently from our approach, they are not interested in finding faults in the variability models, although they also notice that some inaccuracy of their results could be due to some errors in the variability models. Zhang and Becker [30] tackle the problem of managing the variability of the implementation when a variability model is missing: to this purpose, they try to automatically encode variability models by analysing the implementation constraints (i.e., the solution space). Andersen et al. [3] extract feature models from constraints given in CNF or DNF. Since these techniques may introduce some errors, our approach can be used to possibly repair these generated models, or validate their correctness.

Our approach has some similarities with *automatic software repair* [19]. Automatic repair of any software artefact needs some kind of oracle: in our approach the oracle is given by the implementation constraints (i.e., the solution space), while in software repair the oracle is usually given by test suites [6], pre- and post-conditions [24], etc. Moreover, also a way to automatically repair the software is needed: in [18], the authors can try to identify the more effective *repair actions*, that are similar to our repair families.

8 CONCLUSIONS

The paper has presented an approach to automatically repair a variability model that wrongly assesses the validity of some configurations w.r.t. the validity defined by the constraints of the implementation. The approach consists in trying to apply some repairs (in terms of further constraints) to the variability model till it does not consider correct some configurations that cannot be concretized in the implementation. Experiments have shown that the approach is able to repair faulty models used in large and significant projects.

As future work, we plan to devise techniques to avoid the generation of some repairs that will be discarded because of PR or RE. Moreover, we plan to study the effect of our repairs on the corresponding variability model; indeed, it could be that our repairs introduce some redundancies (e.g., redundant constraints in feature models) that make the variability model less readable: we could remove the redundancies on the variability model using exiting tools [29], or we could try to apply the repairs in a way that these redundancies do not occur. Moreover, we plan to devise a technique to assist the user in applying the repair back to the original variability model; indeed, there could be several ways to fix the original model, but some of them could spoil the readability of the model.

ACKNOWLEDGMENT

The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 421–432.

[2] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On Extracting Feature Models from Product Descriptions. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, New York, NY, USA, 45–54.

[3] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. 2012. Efficient synthesis of feature models. In *International Software Product Line Conference*. ACM, 106–115.

[4] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2015. Generating Tests for Detecting Faults in Feature Models. In *ICST 2015 8th IEEE International Conference on Software Testing, Verification and Validation*. 1–10.

[5] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2016. Automatic Detection and Removal of Conformance Faults in Feature Models. In *Software Testing, Verification and Validation (ICST), 2016 IEEE 9th International Conference on*. 1–10.

[6] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Applied Soft Computing* 11, 4 (2011), 3494 – 3514.

[7] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*. Springer-Verlag, Berlin, Heidelberg, 7–20.

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[9] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 491–503.

[10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (Dec 2013), 1611–1640.

[11] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, New York, NY, USA, 173–182.

[13] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. 2011. Reverse Engineering Feature Models from Programs' Feature Sets. In *2011 18th Working Conference on Reverse Engineering*. 308–312.

[14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Towards Automated Testing and Fixing of Re-engineered Feature Models. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1245–1248.

[15] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, New York, NY, USA, 25–32.

[16] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating Consistency between a Feature Model and Its Implementation. In *Safe and Secure Software Reuse: 13th International Conference on Software Reuse, ICSR 2013, Pisa, June 18-20. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[17] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150.

[18] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.* 20, 1 (Feb. 2015), 176–205.

[19] Martin Monperrus. 2015. *Automatic Software Repair: a Bibliography*. Technical Report hal-01206501. University of Lille. http://www.monperrus.net/martin/survey-automatic-repair.pdf

[20] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 140–151.

[21] Sarah Nadi and Richard C. Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. 107–116.

[22] Carlos Parra, Xavier Blanc, and Laurence Duchien. 2009. Context Awareness for Dynamic Service-oriented Product Lines. In *Proceedings of the 13th International Software Product Line Conference (SPLC '09)*. Carnegie Mellon University, Pittsburgh, PA, USA, 131–140.

[23] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wasowski. 2012. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development (FOSD '12)*. ACM, New York, NY, USA, 62–69.

[24] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449.

[25] Matthias Riebisch. 2003. Towards a More Precise Definition of Feature Models. In *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co, Norderstedt, 64–76.

[26] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Computer Networks* 51, 2 (2007), 456 – 479.

[27] Carsten Sinz. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming - CP 2005: 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 827–831.

[28] Paul Temple, José Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Software Product Line Conference (SPLC)*. Beijing, China.

[29] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79, 0 (2014), 70–85.

[30] Bo Zhang and Martin Becker. 2012. Code-based Variability Model Extraction for Software Product Line Improvement. In *Proceedings of the 16th International Software Product Line Conference - Volume 2 (SPLC '12)*. ACM, New York, NY, USA, 91–98.