Combinatorial Interaction Testing for Automated Constraint Repair

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

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

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

***Abstract*—Highly-configurable software systems can be easily adapted to address user's needs. Modelling parameter configurations and their relationships can facilitate software reuse. Combinatorial Interaction Testing (CIT) methods are already often used to drive systematic testing of software system configurations. However, a model of the system's *configurations* not conforming with respect to its software implementation, must be repaired in order to restore conformance. In this paper we extend CIT by devising a new search-based technique able to repair a model composed of a set of constraints among the various software system's parameters. Our technique can be used to detect and fix faults both in the model and in the real software system. Experiments for five real-world systems show that our approach can repair on average 37% of conformance faults. Moreover, we also show it can infer parameter constraints in a large real-world software system, hence it can be used for automated creation of CIT models.**

I. 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. These options, or *features*, can be created during 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 in its category.

Such options can also be decided during compilation time, in order to improve some characteristics of the compiled code (scalability, efficiency, etc.). For example, in 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 and the user wants to switch on/off a particular feature or functionality. 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. The *problem space* described by the model of its configurations shows its supported features and their dependencies, while the *solution space* is the technical realisation of the system [22], i.e., its implementation.

Large configurable systems and software product lines can have hundreds of features. It is infeasible in practice to test all the possible configurations. Consider, for example, a system with only 20 Boolean parameters. One would have to check over one million configurations in order to test them all. Furthermore, the time cost of running one test could range from fraction of a second to hours if not days. In order to address this combinatorial explosion problem, combinatorial interaction testing (CIT) has been proposed for testing configurable systems [7]. It is a very popular black-box testing technique that tests all interactions between any set of t parameters. There have been several studies showing the successful efficacy and efficiency of the approach [17], [18], [25], [26].

Moreover, certain tests could prove to be infeasible to run, because the system being modelled can prohibit certain interactions between parameters. Designers, developers, and testers can greatly benefit from modelling parameters and constraints among them by significantly reducing modelling and testing effort [25] as well as identifying corner cases of the system under test. Constraints play a very important role, since they identify parameter interactions that need not be tested, hence they can significantly reduce the testing effort. Certain constraints are defined to prohibit generation of test configurations under which the system simply should not be able to run. Other constraints can prohibit system configurations that are valid, but need not be tested for other reasons. For example, there's no point in testing the *find* program on an empty file by supplying all possible strings.

Constructing a CIT model of a large software system is a hard, usually manual task. Therefore, discovering constraints among parameters is highly error prone. One might run into the problem of not only producing an incomplete CIT model, but also one that is over-constrained. Even if the CIT model only allows for valid configurations to be generated, it might miss important system faults if one of the constraints is over-restrictive. Moreover, even if the system is not *supposed* to run under certain configurations, if there's a fault, a test suite generated from a CIT model that correctly mimics only desired system behaviour will not find that error. In such situations tests that exercise those corner cases are desirable.

The problem of finding and fixing conformance faults between a given software system and its combinatorial model is a challenging task. Due to the size and complexity of current software systems, the interactions between different software parameters are hard to find. Combinatorial models are frequently derived manually, by expert software engineers.

The Software-artefact Infrastructure Repository[1], for instance, contains models of dozens of software systems, which were derived by hand.

Several methods have been introduced to automate the process of inferring constraints. However, they do not guarantee that the derived model will be correct [29]. Therefore,

we introduce a novel automated approach for finding and fixing conformance faults between the given software system and its combinatorial model.

We use combinatorial interaction testing policies introduced in our previous work [10] to find such faults and fix them by repairing constraints in the original CIT model.

We conduct several experiments aiming to answer the following research questions:

[**RQ1:**] *How effective is our automated constraint repair approach at fixing faults in an existing CIT model?*
In particular, we apply our approach to mutated CIT models and show that our approach can automatically fix on average 37% of conformance faults.

[**RQ2:**] *How effective is our approach in inferring parameter constraints in real-world software?*
We present a case study that shows that our approach can be used to derive constraints for an unconstrained CIT model of a large real-world software system.

[**RQ3:**] *How efficient is our constraint repair approach?*
In order to evaluate the efficiency of our approach, we measure the time taken to repair constraints in CIT models as well as the number of tests needed for the repair. On the five systems evaluated, the mutated models were repaired within seconds.

The paper is structured as follows: Section II gives a brief overview of the field of combinatorial interaction testing; Section II-B briefly describes CIT policies used in our approach; Section III presents definitions and notation used throughout our paper; Section IV presents our approach to fixing conformance faults between a software system and its combinatorial model; Section V contains experimental results and answers to research questions posed; Section VI presents related work and Section VII concludes the paper.

II. Combinatorial Models and Testing of Configurable Systems

Combinatorial Interaction Testing (CIT), often called combinatorial testing or combinatorial testing design, aims to test configurable software systems under the various combinations of its parameter values. There exist several tools and techniques for CIT. Good surveys of ongoing research in CIT can be found in [14], [23], while an introduction to CIT and its efficacy in practice can be found in [19], [25].

A. Combinatorial models and CitLab

A model for a combinatorial problem consists of several parameters (at least 2) which can take various domain values. In most configurable systems, constraints (or dependencies) exist between parameters.

[1]http://sir.unl.edu/portal/index.php

Constraints might be introduced for several reasons, for example, to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply because of design choices [7]. Constraints were first described as being important to combinatorial testing in [5] and were introduced in the AETG system. In our approach tests that do not satisfy the constraints in the CIT model are considered *invalid*.

In this paper we assume that the models are specified using CitLab [4], [11]. This is a framework for combinatorial testing which provides a rich abstract language with precise formal semantics for specifying combinatorial problems, an eclipse-based editor with a rich set of features (syntax highlighting, autocompletion, outline view, and others), and a Java API library which includes utility methods for generating all the test requirements for combinatorial coverage of given strength. CitLab does not have its own test generators, but it relies on other off-the-shelf tools, namely ACTS[2] and CASA[3].

In CitLab parameters and constraints are given in a unique file that contains the whole model. To formally describe a combinatorial problem, the user has to identify at least 2 parameters and their possible values.

Definition 1 (Constraints): Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i can take a value from the domain $D_i = \{v_1^i, \ldots, v_{o_i}^i\}$. Every parameter has a name (it can have also a type with its own name) and every enumerative value has an explicit name. We denote with $Constr = \{c_1, \ldots, c_n\}$ the set of constraints.

Definition 2 (Strength of a test suite): The objective of a CIT test suite is to cover all parameter interactions between any set of t parameters. t is called the strength of the CIT test suite. For example, a pairwise test suite covers all combinations of values between any 2 parameters.

CitLab adopts the language of propositional logic with equality and arithmetic to express constraints. To be more precise, it uses propositional calculus, enriched with the arithmetic over integers and enumerative symbols. As operators, it admits the use of equality and inequality for any variable, the usual Boolean operators for Boolean terms, and the relational and arithmetic operators for numeric terms. CitLab supports also seeds and test goals. However, these are not supported by many tools and they are not considered in this work.

Figure 1 shows the CitLab model of a simple washing machine consisting of 3 parameters. Users can select if the machine has HalfLoad, the desired Rinse, and the speed of Spin cycle. In the Constraints section there are two constraints: if HalfLoad is set then the speed of spin cycle cannot be High; if rinse is set to delicate, then HalfLoad must be true.

B. Combinatorial Testing Policies

In our previous work [10] we proposed to use combinatorial interaction testing techniques to verify the validity of CIT models, which is the approach we use in this work.

[2]http://csrc.nist.gov/groups/SNS/acts/
[3]http://cse.unl.edu/~citportal/

```
Model WashingMachine
Parameters:
 Boolean HalfLoad;
 Enumerative Rinse { Delicate Drain Wool };
 Enumerative Spin { Low Mid High };
end
Constraints:
 # HalfLoad => Spin != Spin.High #
 # Rinse==Rinse.Delicate => HalfLoad #
end
```

Fig. 1: An example CIT model of a washing machine.

In particular, given a CIT model, we modify it according to one of the policies briefly described below. Next, we use one of the standard CIT tools to generate a test suite satisfying the modified CIT model.

We use the term “valid test" to denote the generated configuration that satisfies all the constraints of the original CIT model. Conversely, the term “invalid test" is used for a configuration that does not satisfy at least one of the constraints of the original CIT model. Words “test" and “configuration" are used interchangeably.

In classical combinatorial interaction testing, only valid tests are generated, since the focus is on assessing if the system under test produces valid outputs. However, we believe that invalid tests are also useful. In particular, a combinatorial model can be overconstrained, that is, it might restrict generation of test cases that are valid in the system, yet not be generated due to the constraints in the given model. Furthermore, critical safety systems should be tested if they failed safely in case an invalid configuration is entered. Moreover, creation of a CIT model for a large real-world software system is usually a tedious, error-prone task. Therefore, invalid configurations generated by the model at hand can help reveal constraints within the system under test and help refine the combinatorial model.

Two basic policies can be employed in combinatorial testing: UC (Unconstrained CIT) consists in generating the tests ignoring the constraints, and CC (Constrained CIT) is the classical testing policy that generates only tests satisfying the constraints. Aside from UC and CC, we consider three other policies, introduced in our previous work [10]:

1) Constraints Violating CIT - CV: This approach produces only the tests violating the constraints, i.e. the produced test suite contains every tuple of parameter values that makes at least one constraint false. This approach is complementary with respect to the CC in which only valid configurations are produced.

2) Combinatorial Union - CuCV: CuCV is the union of CC and CV, i.e. it covers all the desired parameter interactions producing valid configurations *and* all those producing invalid ones according to the given CIT model.

3) CIT of Constraint Validity - ValC: ValC requires the interaction of each parameter with the validity of the whole CIT model. That is, both tests that satisfy all the constraints will be generated as well as those that don’t satisfy any of the constraints in the given CIT model. ValC is an approach that tries to balance the validity of the tests without requiring the union of valid and invalid tests.

III. Definitions

We assume that the combinatorial model specifies the parameters and constraints between them for a given software system. We are interested in checking whether this system specification correctly represents the software implementation. We assume that the parameters and their domains are correctly captured in the specification, while the constraints may contain some faults. Software model M belongs to the problem space while implementation of the software system S belongs to the solution space [22].

Formally, given an assignment t that assigns a value to every parameter in P of the model M, we introduce two functions:

Definition 3: Given a model M for a software system S, val_M is the function that checks if assignment t satisfies the constraints in M, while $oracle_S(t)$ checks if t is a valid configuration according to the system S.

We assume that the oracle function $oracle_S$ exists. For instance, in case of a compile-time configurable system, we can assume that the compiler plays the role of an oracle: if the parameter assignment t allows compilation of the product then we say that $oracle_S(t)$ holds. We might enhance the definition of oracle by considering also other factors, for example, if the execution of the test suite completes successfully. However, executing $oracle_S$ might be very time consuming and it might require, in some cases, human intervention.

On the model side, the evaluation of $val_M(t)$ is straightforward, that is, $val_M(t) = c_{1[P \leftarrow t]} \wedge \ldots \wedge c_{n[P \leftarrow t]}$.

Definition 4 (Conformance fault): We say that the constrained CIT model is correct if, for every t, $val_M(t) = oracle_S(t)$. We say that the model contains a *conformance* fault if there exists a t such that $val_M(t) \neq oracle_S(t)$.

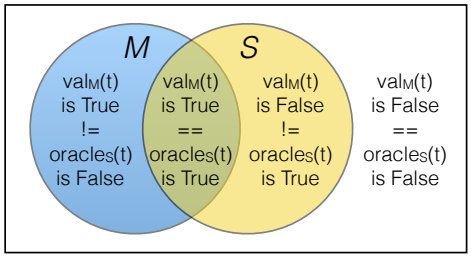


Fig. 2: The space of test cases for system S and its model M. Failing test cases appear in the regions M/S and S/M, i.e., where $val_M(t) \neq oracle_S(t)$.

Figure 2 shows two different types of failing tests with respect to all possible parameter assignments (ignoring constraints). A conformance fault might occur if $val_M(t)$ is $True$ and $oracle_S(t)$ is $False$ (i.e., M/S in Figure 2). Another type of conformance fault occurs, when the model does not pass a test case that is allowed by the system, that is, if $oracle_S(t)$ is $True$, but $val_M(t)$ is $False$ (i.e., S/M in Figure 2).

Example 5: As an example, consider a system S of the washing machine modelled in Figure 1, and as model M the same model in Figure 1 without the first constraint: that conformance fault could be revealed by the following test t:

Parameter Assignments			Function Evaluation	
Rinse	HalfLoad	Spin	$oracle_S$	val_M
Delicate	True	High	False	True

TABLE I: An example test case triggering a conformance fault in the washing machine model.

Definition 6 (Combination): Let P be the set of all parameters in a given model. A combination $comb$ is an assignment of values to every parameter in a (non-empty) subset of P .

Definition 7 (Failure-inducing): A combination $comb$ is failure-inducing if for every test t containing $comb$ (i.e., $comb \subseteq t$), $val_M(t) \neq oracle_S(t)$.

Example 8: Given the System S of the washing machine modelled in Figure 1, and its Model M presented in Figure 1 without the first constraint: a failure-inducing combination in this case would be HalfLoad and Spin=Spin.High, because for every possible configuration t, $val_M(t) \neq oracle_S(t)$, as shown in the table below:

Rinse	HalfLoad	Spin	$oracle_S$	val_M
Delicate	True	High	False	True
Drain	True	High	False	True
Wool	True	High	False	True

TABLE II: All tests containing the failure-inducing combination HalfLoad and Spin=Spin.High.

Definition 9 (Under-constraining and Over-constraining combinations): We say that a combination $comb$ (i.e., a partial assignment) is over-constraining the CIT model if for every assignment t containing $comb$, $val_M(t) = False$ and $oracle_S(t) = True$. We say that a combination $comb$ (i.e., a partial assignment) is under-constraining the CIT model if for every assignment t containing $comb$, $val_M(t) = True$ and $oracle_S(t) = False$.

When a failure-inducing combination is found, one needs to modify the model M, i.e. *repair* the model, so that it faithfully represents the system S that it models. The technique we present in this paper tries to repair the constraints of a combinatorial model whenever a failure-inducing combination is found. There are two types of repairs, depending on the type of the failure-inducing combination:

- If a combination $comb$ is over-constraining, the model must be modified in order to include also the configurations identified by $comb$. This can be done by *relaxing* the constraints by adding $comb$ to each constraint that currently prohibits $comb$ by means of a Boolean *OR* (i.e., $Constr \leftarrow Constr \vee comb$).
- If a combination $comb$ is under-constraining, the model must be modified in order to exclude $comb$ and this can be done by *strengthening* the constraints $Constr$ by adding a further constraint $\neg comb$.

In order to fix the CIT model, we modify its constraints using the failure-inducing combinations found. If $comb$ is over-constraining the CIT model, we build a new constraint by allowing $comb$ to be added: $Constr \leftarrow Constr \vee comb$. If $comb$ is under-constraining, we build a new constraint by adding the negation of $comb$ to the old constraint: $Constr \leftarrow Constr \wedge \neg comb$.

Definition 10 (Relaxing and Constraining Sets): We call a *relaxing set* the set of all combinations *comb* that need to be added to repair the model M. We call a *constraining set* the set of all combinations *comb* that need to be removed to repair the model M.

The aim of our approach is to remove all failure-inducing combinations and thus fix the constraints in the given CIT model. Visually, we want the two circles in Figure 2 to completely overlap, so that there are no more conformance faults between the model and the system.

Example 11 (Repair with a constraining set): Consider the washing machine system S, modelled in Figure 1. Let M be the model presented in Figure 1 without the first constraint: a possible failure-inducing combination is HalfLoad=True, Spin=Spin.High, which is $True$ for val_M and $False$ for $oracle_S$, as shown in Example 8. The constraint "! (HalfLoad & Spin==Spin.High)" would then be added to the constraining set and subsequently to the model M, thus repairing the faulty model.

Example 12 (Repair with a relaxing set): Consider the washing machine system S, modelled in Figure 1. Let M be the model presented in Figure 1 with the first constraint changed to: "HalfLoad => Spin == Spin.Low": a possible failure-inducing combination is "HalfLoad=True, Spin=Mid". The model M would then be corrected by appending to all the constraints, the following combination: " | (HalfLoad & Spin==Spin.Mid)". In this case, the constraint causing the conformance failure is the first one of the model, which after the repair becomes "HalfLoad => Spin==Spin.Low | (HalfLoad & Spin==Spin.Mid)", which is equivalent to "HalfLoad => Spin!=Spin.High". We note here that since we do not use an exhaustive test suite in our approach (details of which are presented in Section IV), the failure-inducing combination derivation might be incorrect. Since we append the same constraint $comb$ to all the constraints, we might potentially need to further repair the model.

In order to measure the faithfulness of the given model M with respect to the system it models S, we introduce the following measure that we call the *failure index*:

Definition 13 (Failure Index): Given a model M of a system S, we define the *failure index* of M to be the number of (valid and invalid) configurations of M that fail. Formally:

$$\mathsf{fi} = \frac{|\{t \in T | val_M(t) \neq oracle_S(t)\}}{|T|},$$

where T is the given test suite.

Taking the exhaustive test suite as T, we can compute an absolute failure index (afi) as measure of the quality of a model. However, computing the absolute failure index afi is infeasible in general. In the experiments, we will be able to compute the models afi by assuming that we know the exact model of the system. We use multi-valued decision diagrams (MDDs) [12] in order to calculate the number of tests that are permitted by the constraints in the model.

IV. The constraint repair process

In this section we present the proposed process for constraint repair in CIT models. In our previous work [10] we devised a way of finding conformance faults. We use these techniques to find faults and extend them by proposing an automated way of fixing the faults found. Figure 3 presents an overview of the constraint repair process.

We first generate a set of test cases of given strength k based on one of the CIT policies described in Section II-B. We evaluate each of the tests t using the function $val_M(t) = oracle_S(t)$. We mark each test for which $val_M(t) \neq oracle_S(t)$ as a failing one (since it reveals a conformance fault). We say that the test t passes, otherwise.

We input the generated test suite and the result of $val_M(t) = oracle_S(t)$ for each test t into a combinatorial testing-based fault localisation tool called BEN [13]. BEN uses a heuristic approach to identify a set of failure-inducing combinations of given size (i.e., combinatorial strength). Given an initial test suite and the result for each test, it identifies a set of suspicious parameter-value combinations. Next, it proceeds in an iterative manner: it generates a set of new tests and queries the user for the result of these tests; after the user supplies the data, BEN identifies a new set of suspicious combinations and a new set of tests; the process is repeated until a set of failure-inducing combinations is found or all tests pass (in which case we can increase the strength of CIT testing)[4]. Further details about the heuristic approach implemented in BEN can be found in the following paper: [13]. We note that the problem of finding minimum failure-inducing combinations is a challenging task, since generation of an exhaustive test suite is often infeasible.

Once failure-inducing combinations are found, we modify the constraints of the original CIT model, as explained in Examples 11 and 12. We repeat the whole process until all test cases (generated by the CIT policy of choice and BEN) pass.

Our proposed constraint repair process proceeds as follows:

1) Start with a constrained CIT model containing a set of constraints C.
2) Derive a t-way CIT test suite according to one of the CIT policies described in Section II-B.
3) If for all test cases, the test result is the same according to the model and according to the system, then exit.
4) For all tests t such that $val_M(t) \neq oracle_S(t)$, mark t as a failing test. t is passing otherwise. Use BEN to derive failure-inducing combinations:
   a) Produce an initial set of suspicious combinations with new test cases using BEN.
   b) Add tests produced by BEN to the test suite.
   c) Mark the new test cases as failing or passing according to $val_M = oracle_S$.
      i) If all new tests pass and BEN has not detected any failure-inducing combinations, increase the test suite strength.
      ii) Otherwise, input the new set of tests to BEN.
   d) If BEN terminates and produces a set of failure-inducing combinations, exit BEN.
5) Modify the constraint set C based on the result produced by BEN:
   Given the set of failure-inducing combinations $comb$s, for each $comb$:
   a) If a failure-inducing configuration $comb$ occurs in test cases for which $val_M(t) = True$ and $oracle_S(t) = False$ (i.e., belongs to the constraining set), then $Constr \leftarrow Constr \cup \neg comb$.
   b) If a failure-inducing configuration $comb$ occurs in test cases for which $val_M(t) = False$ and $oracle_S(t) = True$ (i.e., belongs to the relaxing set), then for each $c_i \in Constr$, add $comb$ to c_i, that is, $c_i \leftarrow c_i \vee comb$.
6) Go back to point 1.

Note that this approach can be used to infer constraints by supplying an unconstrained CIT model to our tool.

The final constraints produced by our process can be further simplified. Since constraints in our model can be represented in Boolean form, we can use propositional logic rules. We leave this step as future work.

Another enhancement would be identification of constraints to which $comb$s from the relaxing set need to be added. However, identification of the set of constraints that violate a $comb$ is a non-trivial task. It can be reduced to the problem of finding minimum unsatisfiable sets in Boolean satisfiability solving (SAT). We leave this step as future work.

V. Experiments

In order to test our approach we conducted the following two experiments[5]. In the first experiment, we applied mutation to a set of models taken from the literature. In the second experiment, we used a configurable software system, namely Django, in order to test our framework on a real case study.

A. Mutation Analysis

We gathered a set of combinatorial models taken from several papers and applied our process to mutated versions

[4]In our experiments we also found there are other cases when BEN terminates. These are, however, usually error states. We treat these cases the same way in which we deal with the 'all test pass' case, that is, we report that BEN did not find any failure-inducing combinations of given strength.

[5]All the experiments have been executed on a Linux PC with two Intel(R) i7-3930K CPU (3.2 GHz) and 16 GB of RAM.

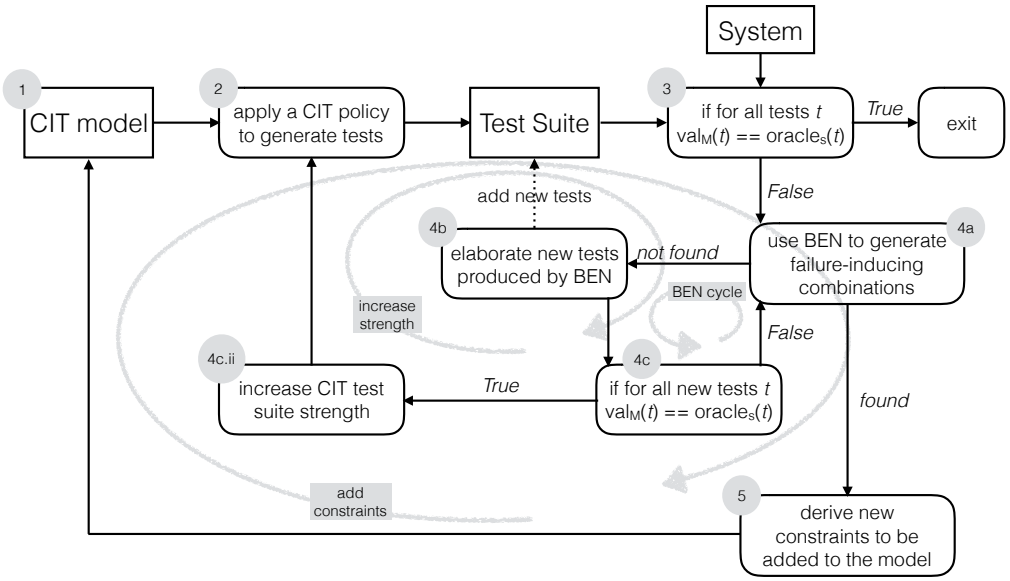


Fig. 3: The constraint repair process.

of these models. In particular, for every model m in the benchmarks we derived a set M of mutants.

We have applied simple mutation operators to the constraints in the model under test in order to find mutants to be repaired. Our mutation operators are of 4 types: add a constraint that excludes a value of a parameter, remove an entire constraint, negate a constraint, and change a logical operator to another one (*AND* to *OR* etc.). Then, for each mutant m' in M, representing a possible faulty model, we applied our repair process by using the original model m as the oracle (to compute the $oracle_S$ function).

We considered all CIT policies presented Section II-B. We applied our repair starting with combinatorial strength set to 1 and increase it up to 3 (i.e., we are only concerned with finding failure-inducing combinations of size 1,2 and 3). We included combinatorial tests of strength 1 since some faults can be already detected by a single parameter assignment.

B. Benchmarks

We used 5 case studies to evaluate our proposed approach:

1) The **WashingMachine** system, model of which is presented in Figure 1. It represents an abstract view of the human interaction with an *embedded system*.

2) **Concurrency** is a testing problem for real-life concurrent system presented in [27].

3) **Telecom** is a real-life telecommunication system presented in [27].

4) **Aircraft** is a small Software Product Line (SPL) found in SPLOT repository and presented in [30].

5) **Libssh** is the combinatorial model for cmake of SSH library taken from [2].

Table III presents various benchmark data: number of variables, number of constraints (including the number of clauses when converted to conjunctive normal form (CNF)), the size of the state space (the total number of possible configurations), and the percentage of configurations that are valid (i.e. the ratio of valid configurations). Note that a low ratio indicates that there are only few valid configurations. We tried to collect models from different domains, with a good level of diversity (in terms of size, constraints, and so on) in order to increase the validity of our findings.

TABLE III: The benchmark data (for CNF size a^b means b clauses with a literals each.)

name	#var	constraints #	constraints CNF size	State space exp	State space #conf	validity ratio	mutants
WashingM.	3	2	2^3	$2^1 3^2$	18	66%	18
Concurrency	5	7	$2^4 3^1 5^2$	2^5	32	25%	38
Aircraft	8	2	$3^1 4^1$	$2^7 3^1$	384	82%	25
Libssh	16	2	2^2	2^{16}	65536	50%	42
Telecom	10	21	$2^{11} 3^1 4^9$	$2^5 3^1 4^2 5^1 6^1$	46080	39%	116

We ran each constraint repair process 10 times and report averages. We used the ACTS tool for CIT test suite generation.

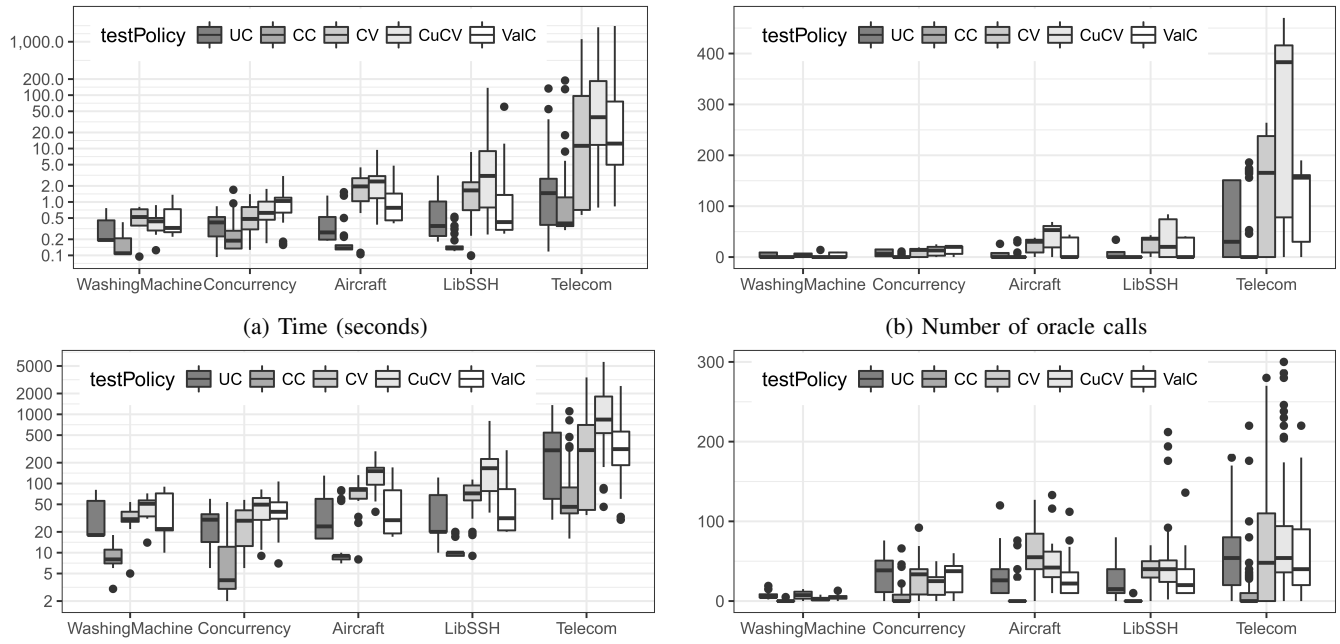
Table III reports the number of mutants we generated for each model.

Evaluation of Effort and Repair Capability by mutation: In order to evaluate our technique, we wanted first to measure the required *effort* and how it varies by changing the CIT policy. To measure the effort we used, for each mutant:

Time The total time required to fix a mutated model, including test generation, BEN invocation, and oracle evaluation. The time required by the oracle to evaluate a test is negligible in this experiment, since we use the original model as an oracle.

Tests The number of test cases are generated according to the given testing policy. Note that for a mutant, test case generation can be invoked several times.

This can happen if, e.g., not all the tests pass and BEN



(a) Time (seconds)

(b) Number of oracle calls

(c) Tests

(d) BEN (additional tests)

Fig. 4: Effort of the repair process

is not able to find the failure-inducing combination and we have to increase the combinatorial testing strength.

Oracle The number of times the oracle was called. Since evaluation of the function $oracle_S$ can be expensive, especially if some human input is required, this is a critical factor.

BEN The number of additional tests BEN requires in order to isolate the failure-inducing combinations.

We are also interested in assessing the *repair capability* of our approach. We use the following two measures:

TRM The ratio of completely repaired models. In this experiment, we can check if a repaired model is completely fault free by querying whether it is equivalent to the original model. To check equivalence among combinatorial models we use an SMT solver [1].

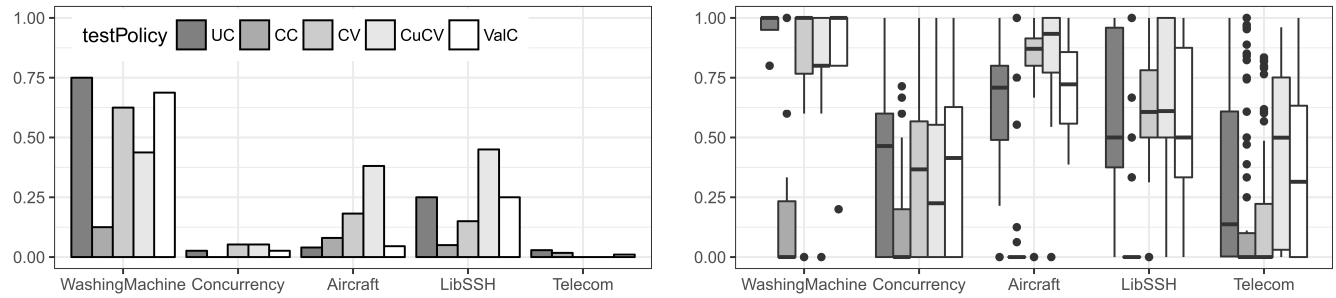
FID Even if a model is not completely repaired, we are interested in measuring how many conformance faults were repaired. We introduce the failure index delta, as **FID** $= (\mathsf{afi}_{init} - \mathsf{afi}_{final})/\mathsf{afi}_{init}$ which is defined in terms of the absolute failure index presented in Def. 13. **FID** represents the percentage of conformance faults that are repaired.

Figures 4 and 5 show respectively the effort and the repair capability of each policy for every model in the benchmark set. The experimental results are also summarized in Table IV. We notice that in terms of time (Fig. 4a), our process is rather fast for small models, but also for the biggest models, like Telecom, it terminates on average in less than 10 seconds for weak policies (like UC and CC), and in any case in less than 10 minutes when using strong policies like CuCV and ValC. In terms of number of tests, we can observe that the number of tests generated at the beginning of each repair process (Fig. 4c) and the number of tests BEN requires in order to find failure-inducing combinations (Fig. 4d) are correlated. If one starts with small test suites, BEN will require fewer tests (and it likely identifies fewer failure-inducing combos). Regarding the repair capability, the TRM is rather small for big specifications, and using more powerful testing policy (like CuCV) does not help much (Fig. 5a). Using our process to obtain a completely bug-free model seems unrealistic for big models: CIT likely provides an insufficient coverage for this. However, if we consider the FID in Fig. 5b, we can say that almost always our technique and CIT improve the model except when using CC. The overall minimum average for **FID** when using strong policies like CuCV and ValC is 29%. For big models, there are cases in which we are unable to improve the constraints, but on average we can still remove around 35% of the faults with CuCV and ValC. As already observed in [10], the CC policy (that generates only valid tests) detects fewer conformance faults, and for this reason has the lowest FID. The average FID over all the mutants is around 37%. The average FID when using CuCV is 49%: half of faults can be repaired on average if the user chooses this policy.

C. Repairing the model of Django parameters

Django is a free and open source web application framework, written in Python, that supports the creation of complex, database-driven websites, emphasizing reusability of components[6]. Each Django project has a configuration file.
It is loaded at *launch time*, i.e. every time the web server that executes the project (e.g. Apache) is started. Among all the

[6]https://www.djangoproject.com/



(a) TRM: Totally repaired models (over all the mutations)

(b) FID: Failure Index Delta

Fig. 5: Repair capability

TABLE IV: Means of the quantities over all the mutations

	Time (seconds)					Tests (Initial)					BEN (additional tests)					FID (%)				
name	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC
WashingM.	0.3	0.2	0.5	0.4	0.5	31.5	9.6	29.7	45.0	36.3	6.6	0.6	6.5	2.5	4.6	95	21	81	82	86
Concurrency	0.4	0.3	0.6	0.8	1.1	27.8	11.3	29.1	47.0	41.9	33.0	8.0	29.9	20.7	30.8	36	13	33	29	36
Aircraft	0.4	0.3	2.0	2.9	1.3	41.1	18.1	74.0	144.5	51.6	33.3	8.6	59.5	50.4	31.5	62	14	79	85	60
Libssh	0.7	0.2	1.8	77.9	2.6	44.0	10.8	66.5	187.5	54.6	23.4	1.0	35.3	46.3	26.0	57	10	63	67	50
Telecom	4.5	4.0	83.0	180.3	107.7	352.1	111.8	532.1	1287.0	435.9	56.1	11.5	70.0	77.6	60.1	25	16	12	35	31

possible configuration parameters, we selected and modelled one Enumerative and 23 Boolean parameters. We implemented an automated oracle which returns true if and only if the HTTP response code of the Django project homepage is 200 (HTTP OK). This oracle invocation is costly in terms of time, since editing the settings of the Django project, starting the Apache server, and waiting for its response, requires around 6 seconds.

We applied the constraint repair process described in Section IV on a default Django start project running on Apache server (without SSL configuration) called at address *localhost*. We started from a model (Django0) with an empty set of constraints because we thought that Django was accepting (and amending if necessary) every possible configuration, and secondly, to test the effectiveness of this technique to *infer* (and not only *repair*) configuration constraints: we call this first experiment "Inference from Django0". At the end of the process, we obtained different set of constraints depending on the policy. For all the policies, we executed the repair process with test suite strength from 1 to 3.

We noticed that the application of the testing policy CV produces an empty test suite at the beginning, and an empty set of constraints at the end of the process, because the initial Django model contains no constraints, and in particular there are no test cases violating constraints. So we ignored this policy in the experiments.

We manually derived the constraints, based on the Django documentation about configuration files, and by looking at the results of a few test cases. As a second experiment, we applied the constraint repair process to this manual model (Manual), in order to improve its conformance with the real Django system: we call this second experiment "Repairing Manual".

TABLE V: Django inference and repair results

	Policy	Tests	BEN	Oracle	Time	C[a]	P[b]	fi
Django0 (empty model)						0	0	0.789
inference from Django0	UC	36	20	32	201s	5	6	0.325
	CC	173	60	154	1671s	8	9	0.254
	CuCV	320	54	335	3570s	9	12	0.081
	ValC	288	80	124	1453s	17	19	0.204
Manual						3	4	0.033
repairing Manual	UC	63	30	51	435s	3	4	0.033
	CC	63	30	70	436s	3	4	0.033
	CuCV	202	30	167	1354s	4	4	0
	ValC	118	30	92	564s	4	4	0

[a]C: Total number of constraints in the model

[b]P: Total number of parameters involved in the constraints (without considering duplicates)

Results of this experiment are summarized in Table V, which reports the number of tests generated by the testing policy, the number of additional tests required by BEN, the time taken for the whole process, the number of constraints of the final model, the number of parameters involved in the constraints, and the failure index. To compare the different policies in terms of fault detection capability, in this case, we cannot rely on the absolute failure index, since the actual model of the Django system remains unknown. We can, however, compute the failure index (see Definition 13) where the test suite T is the union of all the test suites we generated for all the policies.

With regards to *inferring* constraints, the CuCV policy generates the biggest test suite and requires also the largest amount of time: around 1h for constraint inference from Django0 (including oracle invocation). It has also the lowest fi (8%), which however is not zero: no inferred model is completely fault-free. The UC policy is the worst in terms

of failure index data, but it is the fastest in our experiment. ValC produces a model with the largest number of constraints and the failure index (20%) is the second lowest. However, if we compare the failure index of Diango0 (79%), we can say that all the testing policies can substantially improve the initial empty model. In conclusion, all the models contain a reasonable number of constraints and although our best policy (CuCV) does not beat the manual model we have developed (by using our best efforts), it can produce a rather correct model.

Regarding the problem of *repairing* the initial manual model, two out of four testing policies (CuCV and ValC) are able to completely repair the manual model (at least when considering the failure index based on the set of generated tests), while the UC and CC policies did not improve the initial model. For constraint repair, the number of tests and the time required is generally lower (except for UC) than for the process of inferring constraints. We conclude that the availability of an initial model, even if it is not completely correct, makes possible to obtain a fault-free model for a real system when strong testing policies are applied.

Figure 6 reports the constraints for Django0 (the initial model with empty set of constraints), Manual (the model *inferred* manually), and for the repaired models produced using CuCV and ValC (starting from Manual). The constraints obtained in the final models are still readable, making a human inspection possible for further modifications.

```
Django0
# true #

Manual (constraints "inferred "  manually)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
// if not DEBUG, "localhost" must be allowed
# !DEBUG =>
    (ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST) #

CuCV (repairing Manual)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
# !DEBUG =>
  ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST #
# ALLOWED_HOSTS==IP =>
    !DEBUG or PREPEND_WWW or SECURE_SSL_REDIRECT #

ValC (repairing  Manual)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
# !DEBUG =>
    ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST #
# ALLOWED_HOSTS==IP => !DEBUG or SECURE_SSL_REDIRECT #
```

Fig. 6: Django models obtained by repairing the manual model using different testing policies

VI. Related Work

The problem of finding and fixing constraints in combinatorial models has been addressed in the field of software product lines. Henard et al. [16] assumes existence of valid configurations of the given software system and use a SAT solver to randomly generate configurations for an existing feature model of the system. Once a conformance fault is found, a constraint is added, removed or altered with certain probability. The mutated feature model is compared against the original using a pre-defined fitness function and the best of the two is kept for future evaluation.

In contrast to Henard et al.'s work, we derive test cases in a systematic fashion using combinatorial interaction testing. Moreover, by applying policies presented in [10], we are able to generate invalid configurations. Therefore, in contrast to the work by Henard et al., we are able to find conformance faults caused by an over-constrained model. Finally, we use a systematic approach to fix the model by utilising BEN to find the minimum fault-inducing configurations.

Arcaini et al. [2] used mutation testing to detect and fix conformance faults by distinguishing a given feature model from its mutants.

The problem of modelling and testing the configurations of complex software systems is non-trivial. There has been much research done in extracting constraints among parameter configurations from real systems. For instance, the importance of having a model of variability and having the constraints in the model aligned with the implementation is discussed in [22]. However, in that paper, authors try to identify the source of configuration constraints and to automatically extract the variability model. Our approach is oriented towards the validation of a variability model that already exists. Moreover, they target C-based systems that realise configurability with their build system and the C preprocessor. A similar approach is presented in [28], where the authors extract the compile-time configurability from its various implementation sources and examine for inconsistencies (for example, dead features and infeasible options). We believe that our approach is more general (not only compile-time and C-code) and can be complementary in validating and improving automatically extracted models.

Testing configurable systems in the presence of constraints is tackled in [6] and [25]. In these papers, authors argue that CIT is a very efficient technique and that constraints among parameters should be taken into account in order to generate only valid configurations. This allows to reduce the cost of testing. Also in [3], authors showed how to successfully deal with constraints by solving them using a constraint solver such as a Boolean satisfiability solver (SAT). However, the emphasis of that research is more on testing of the final system not its combinatorial model. CIT is also widely used to test SPLs [24].

In SPL the validation and extraction of constraints between features is generally given in terms of feature models (FMs). Synthesis of FMs can be performed by identifying patterns among features in products and in invalid configurations and build hierarchies and constraints (in limited form) among them. For instance, Davril et al. applied feature mining and feature associations mining to informal product descriptions [8]. There exist several papers that apply search based techniques, which generally give better results [9], [15], [20], [21]. However, checking and maintaining the consistency between a SPL and its feature model is still an open problem.

VII. Conclusions

We propose a novel approach for automatically finding and fixing faults in models of parameter configurations of software systems. In particular, we described how combinatorial testing techniques can be utilised for this purpose.

We used novel CIT policies introduced in our previous work [10] that can help software testers discover faults in the model of system configurations as well as faults in the software implementation that the model describes. We call these conformance faults.

In this paper we present a process for finding and fixing conformance faults. We conduct several experiments on five software systems to validate our approach. We show that we can successfully repair existing combinatorial models. Furthermore, we also show how our approach can be utilised to derive constraints between parameters of large complex software systems. The technique presented should help software developers derive and fix combinatorial testing models by automating this often purely manual and thus time-consuming and highly error-prone task.

Acknowledgements

We would like to thank Laleh Sh. Ghandehari for her assistance in the use of BEN and Paolo Vavassori for help with the experiments.

References

[1] P. Arcaini, A. Gargantini, and P. Vavassori. Validation of models and tests for constrained combinatorial interaction testing. In *The 3rd International Workshop on Combinatorial Testing (IWCT 2014) In conjunction with International Conference on Software Testing ICSTW*, pages 98–107. IEEE, 2014.

[2] P. Arcaini, A. Gargantini, and P. Vavassori. Automatic detection and removal of conformance faults in feature models. In *Software Testing, Verification and Validation (ICST), 2016 IEEE 9th International Conference on*, April 2016.

[3] A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. Springer.

[4] A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tool track*, 2013.

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7):437–444, 1997.

[6] M. Cohen, M. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Trans. on*, 34(5):633–650, 2008.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.

[8] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Clelang-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions, Aug. 22 2013.

[9] J. M. Ferreira, S. R. Vergilio, and M. A. Quináiaferreia. A mutation approach to feature testing of software product lines. In *The 25th International Conference on Software Engineering and Knowledge (SEKE) Engineering, Boston, MA, USA, June 27-29, 2013*, pages 232–237. Knowledge Systems Institute Graduate School, 2013.

[10] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 49–63, 2016.

[11] A. Gargantini and P. Vavassori. CitLab: a laboratory for combinatorial interaction testing. In *Workshop on Combinatorial Testing (CT) In conjunction with International Conference on Software Testing (ICST 2012, April 17-21)*, pages 559–568, Montreal, Canada, 2012.

[12] A. Gargantini and P. Vavassori. Efficient combinatorial test generation based on multivalued decision diagrams. In E. Yahav, editor, *Hardware and Software: Verification and Testing, Haifa Verification Conference HVC 2014*, volume 8855 of *Lecture Notes in Computer Science*, pages 220–235. Springer International Publishing, 2014.

[13] L. S. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. BEN: A combinatorial testing-based fault localization tool. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4. IEEE, 2015.

[14] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[15] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 5–18, New York, NY, USA, 2014. ACM.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Towards automated testing and fixing of re-engineered feature models. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1245–1248, 2013.

[17] D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. In *SEW '06: IEEE/NASA Software Engineering Workshop*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[18] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.

[19] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[20] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Search Based Software Engineering*, pages 168–182. Springer, 2012.

[21] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and Software Technology*, 61:33 – 51, 2015.

[22] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: static analyses and empirical results. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *ICSE*, pages 140–151. ACM, 2014.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.

[24] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proc. of the International Conference on Software Testing (ICST)*, pages 459–468, Paris, France, April 2010. IEEE.

[25] J. Petke, M. B. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, 2015.

[26] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 26–36, 2013.

[27] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 254–264, New York, NY, USA, 2011. ACM.

[28] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.

[29] P. Temple, J. A. G. Duarte, M. Acher, and J.-M. Jézéquel. Using machine learning to infer constraints for product lines. In *Software Product Line Conference (SPLC)*, 2016.

[30] M. Voelter. Using domain specific languages for product line engineering. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 329–329, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.