DOI 10.1109/QRS-C.2019.00045

Repairing Over-Constrained Models for Combinatorial Robustness Testing

Konrad Fögen
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
foegen@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Testing negative scenarios is important to evaluate robustness of software systems. Error-handling can terminate the system before all values are evaluated and faults can remain undetected. Therefore, extensions for combinatorial testing separate generation of positive and negative scenarios. Unfortunately, it is easy to create over-constrained models. Certain values or value combinations are prevented from appearing in the test suite and remain untested. In this paper, we define over-constrained models and present a technique to identify and repair them.**

***Index Terms*—Robustness Testing, Combinatorial Testing**

I. Introduction

Robustness is an important characteristic of software systems which describes “the degree to which a system or component can function correctly” in the presence of invalid inputs [1], e.g invalid values like a string value when a numerical value is expected, or invalid value combinations like a begin date which is after the end date. To improve robustness, error-handlers are implemented to appropriately react to external faults. Unfortunately, error-handlers can contain up to three times more faults than normal code [2]. Therefore, the behaviour of a system under test (SUT) should be tested in both positive and negative scenarios.

Combinatorial testing (CT) is a well-known black-box testing approach [3]. Based on an input parameter model (IPM) with parameters and values, the objective is to generate a set of test inputs of that satisfies a combinatorial coverage criterion.

In CT, input masking is a phenomenon which the tester must take care of [4]. Test inputs can lead to input masking if they contain at least one invalid value or invalid value combination. Once the SUT starts evaluating an invalid value, the SUT is expected to initiate error-handling by switching from the normal to the exceptional control-flow and to respond with an error message. The remaining values and value combinations of the test input remain untested as they are *masked*.

To avoid input masking, combinatorial robustness testing is an extension where valid and invalid test inputs are distinguished [5]–[8]. Semantic information is added to the IPM to recognize invalid values and invalid value combinations.

Some CT tools include the concept of invalid values to divide values into two disjoint subsets: $V_i = V_i^{valid} \cup V_i^{invalid}$. But, invalid value combinations are not directly considered. To model them, a workaround is required [7]. In contrast, an approach we proposed in previous work [7], [8] directly considers invalid value combinations. Invalid values and invalid value combinations are described as so-called error-constraints. A value or value combination is valid if it satisfies all error-constraints. If not, it is invalid.

A combination strategy generates valid test inputs that do not contain any invalid values and invalid value combinations. Invalid test inputs are generated such that each invalid value and invalid value combination appears in separate test inputs.

While the approaches with additional semantic information work in general, it is easy to create over-constrained models when applying it in practice. As a consequence, not all specified invalid values and invalid value combinations appear in the test inputs and faults could remain undetected.

In this paper, we therefore define over-constrained models and present techniques to identify and explain them.

The paper is structured as follows. First, an example to illustrate the problem of over-constrained models is presented. Section III and IV summarize foundations and related work. In Section V, the concept of over-constrained models is defined and techniques for identification and explanations are discussed. Afterwards, experiments are presented. We conclude with a summary of our work.

II. Example

Throughout the paper, we use a customer registration service as an example. To ensure data quality, the service has to check that the entered data actually matches the intended semantics of the input fields. The following checks have to be done: Empty inputs should be avoided, a person’s title should match the given name and the given name should not be interchanged with the family name. Since the service cannot correct wrong data itself, it should return an error message asking the user to correct the data.

An IPM for the customer registration example is depicted in Figure 1. Invalid values and value combinations are modelled via error-constraints describing invalid values like `[GivenName:123]` and invalid value combinations like `[Title:Mrs, GivenName:John]`. The test input `[Title:Mrs, GivenName:Jane, Family Name:123]` is invalid because it contains a family name that does not satisfy error-constraint c_3. It is also an example for invalid input

$p_1 : Title$	$V_1 = \{Mr, Mrs, 123\}$
$p_2 : GivenName$	$V_2 = \{John, Jane, 123\}$
$p_3 : FamilyName$	$V_3 = \{Doe, 123\}$

$c_1 : Title \neq 123$
$c_2 : GivenName \neq 123$
$c_3 : FamilyName \neq 123$
$c_4 : Title = Mrs \Rightarrow GivenName \neq John$
$c_5 : Title = Mr \Rightarrow GivenName \neq Jane$

Figure 1. Exemplary IPM with Five Error-Constraints

masking. The family name leads to error-handling which returns an error message to the user *before* the values of the other parameters are evaluated.

To prevent input masking, a combination strategy generates valid test inputs which satisfy all error-constraints and according to a given combinatorial coverage criterion [7]. Afterwards, invalid test inputs are generated according to another combinatorial coverage criterion. Depending on the coverage criterion, each modelled invalid value and invalid value combination should appear in at least one test inputs of which all other values and value combinations are valid.

For the given example, at least one test input must contain `[Title:123]` as specified by error-constraint c_1 which satisfies all other error-constraints. Another example, at least one test input must contain `[Title:Mrs, GivenName:John]` as specified by c_4 which satisfies all other error-constraints.

To make the model over-constrained, consider a modified example with a slightly rewritten error-constraint c'_4 that uses the $=$ operator instead of $\neq$.

$$c'_4 : Title = Mrs \Rightarrow GivenName = Jane$$

The error-constraint is semantically equal for the generation of valid test inputs but has implications for the generation of invalid test inputs. In addition to `[Title:Mrs, GivenName: John]`, the modified error-constraint c'_4 also describes the invalid value combination `[Title:Mrs, GivenName:123]`. Therefore, at least two invalid test inputs must be generated for both invalid value combinations. However, a combination strategy cannot find a test input that contains `[Title:Mrs, GivenName:123]` and satisfies all other error-constraints at the same time because of a conflict with error-constraint c_2.

This is one example of a conflict caused by an over-constrained model. As a consequence, not all specified invalid values or invalid value combinations appear in the test inputs and faults could remain undetected.

III. Background

In this section, we briefly discuss foundations of robustness testing, CT and combinatorial robustness testing. For more detailed explanation, please refer to previous work [8].

A. Robustness Testing

According to IEEE [1], "the degree to which a system or component can function correctly" in the presence of external faults, is called **robustness**. External faults are faults in the environment of a system like invalid inputs from a user or stressful environmental conditions which causes third-party services to not respond in time. To make a system robust, the system must become fault tolerant with regards to external faults by considering, specifying and implementing error-handlers. Unfortunately, the error-handlers can contain three times more (internal) faults than normal code [2].

Therefore, a system under test (SUT) should be tested with both positive and negative scenarios. Positive scenarios focus on valid intended operations of the SUT using valid input values that are within the specified boundaries. Negative scenarios focus on the error-handling using invalid values and invalid value combinations as inputs.

B. Combinatorial Testing

1) Overview: Combinatorial testing (CT) is a black-box test design technique that systematically generates test inputs based on a given **input parameter model** (IPM) with the objective to satisfy a combinatorial coverage criterion. The IPM is represented as a set of n input parameters $IPM = \{p_1, ..., p_n\}$ and each input parameter p_i is represented as a non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$. A **tuple** is a set of parameter-value pairs for d distinct parameters such as `[Title:Mr, GivenName:John]`. A tuple with n parameter-value pairs of is a **test input** which can be used to stimulate the SUT. A tuple τ_a **covers** another tuple τ_b if and only if all parameter-value pairs of τ_b are contained in τ_a.

2) Input Masking Effect: In CT, input masking is a phenomenon which the tester must take care of [4]: "The input masking effect is an effect that prevents a test case from testing all combinations of input values, which the test case is normally expected to test". Here, input masking is caused by test inputs that contain certain values or value combinations. Because of these values or value combinations, the test input is not or only partially evaluated by the SUT. The other values and value combinations of the test input remain untested as long as there is no other test input that includes them.

3) Irrelevant Value Combinations: Most real-world systems have restrictions in their input domains and most IPMs contain combinations of parameter values that should not be combined [9]. These value combinations are **irrelevant** as they are, for instance, not executable or just not of any interest. For instance, a configuration like `[Browser:Edge, OS:Linux]` cannot be executed. Since it cannot be executed, the other values and value combinations remain untested if there is no other test input that covers them. Irrelevant value combinations are one cause of the input masking effect. Irrelevant value combinations *mask* all other values and value combinations of the same test input. Therefore, irrelevant value combinations should be detected and removed from the test suite.

Constraint handling is one strategy to avoid irrelevant value combinations in test inputs while still preserving the coverage criterion [9]. Constraints are explicitly modeled as logical expressions that describe conditions [10]. A function $\Gamma(\tau, C) \rightarrow$ `Bool` evaluates whether a tuple τ satisfies a set of constraints C. A combination strategy generates test inputs that satisfy the constraints and excludes irrelevant value combinations. We denote a set of constraints to distinguish between relevant and irrelevant tuples as **exclusion-constraints** (C^{ex}).

DOI 10.1109/QRS-C.2019.00045

A tuple τ is **relevant** if it satisfies every exclusion-constraint: $\Gamma(\tau, C^{ex}) = \texttt{true}$. A tuple is **irrelevant** if at least one exclusion-constraint remains unsatisfied: $\Gamma(\tau, C^{ex}) = \texttt{false}$.

C. Combinatorial Robustness Testing

1) Overview: Robust systems process valid inputs and contain error-handlers to deal with invalid inputs by users or third-party systems. The first invalid value or invalid value combination that is evaluated by a SUT is expected to trigger error-handling and the normal control-flow is left. Broadly speaking, the error-handler responds with an error-message and terminates. Invalid values and invalid value combinations are another cause of the input masking effect. Once error-handling is triggered by an invalid value or invalid value combination, all other values and value combinations of the test input remain untested as they are masked.

Therefore, relevant tuples can be further partitioned into valid and invalid tuples. **Valid** tuples are relevant and do not contain any invalid value or invalid value combinations to prevent error-handling. **Invalid** tuples are relevant but contain at least one invalid value or one invalid value combination to trigger error-handling. A **strong invalid** tuple is relevant and contains exactly one invalid value or exactly one invalid value combination to prevent that one masks the other.

Valid test inputs are generated such that they satisfy a combinatorial coverage criterion like t-wise coverage excluding all irrelevant and invalid values and value combinations. Invalid test inputs are generated satisfying another coverage criterion such as **single error coverage** [3] which is satisfied if each invalid value and each invalid value combination appears in at least one test input of which all other values are valid or the **single error t-wise coverage** [6], [7] where each invalid value and each invalid value combination is combined with all valid value combinations of t parameters.

2) Generating Strong Invalid Test Inputs: Two different approaches for combinatorial robustness testing exist. The first approach is implemented by CT tools like AETG [5], ACTS [11] and PICT [6]. They include the concept of invalid values to generate invalid test inputs that satisfy single error coverage and single error t-wise coverage. The values are divided into two disjoint subsets to represent valid and invalid values, i.e. $V_i = V_i^{valid} \cup V_i^{invalid}$. The invalid values of each parameter are excluded from the generation such that the test inputs satisfy the t-wise coverage criterion for the valid parameter values. Afterwards, each invalid value is combined with valid values of the other parameters such that every test input contains exactly one invalid value. Invalid value combinations are not directly supported. A combination of invalid values and exclusion-constraints is required as a workaround [7].

The second approach was introduced by us to directly support invalid value combinations [7], [8]. Instead of distinguishing valid and invalid values, we introduce a new group of constraints to describe invalid values and invalid value combinations: **error-constraints** (denoted as C^{err}).

Again, a tuple τ is relevant if it satisfies all exclusion-constraints. A relevant tuple is **valid** if all exclusion-constraints

```
input:  IPM, t, $C^{ex}$, $C^{err}$
output: A set of test cases
    let V = { $V_1$, ..., $V_i$ }
  let $T^+$ = $gen^+$(V, t, $C^{ex} \cup C^{err}$)
  let $T^-$ = $\emptyset$
  foreach $c_i$ in $C^{err}$
    let $\overline{c_i}$ = negation of $c_i$
    let C' = ($C^{err} \setminus \{c_i\}$) $\cup$ {$\overline{c_i}$}
    $T^-$ = $T^-$ $\cup$ $gen^-$(V, t, $C^{ex}$ $\cup$ C')
  return $T^+ \cup T^-$
```

Listing 1. Combination Strategy with Error-Constraints

are satisfied and if all error-constraints are satisfied as well: $\Gamma(\tau, C^{err} \cup C^{ex}) = \texttt{true}$. A relevant tuple is **invalid** if all exclusion-constraints are satisfied but at least one error-constraint remains unsatisfied: $\Gamma(\tau, C^{ex}) = \texttt{true}$ and $\Gamma(\tau, C^{err}) = \texttt{false}$. An invalid tuple τ is **strong invalid** if and only if exactly one error-constraint remains unsatisfied: $\exists! c \in C^{err} : \Gamma(\tau, \{c\}) = \texttt{false}$ and $\Gamma(\tau, C^{err} \setminus \{c\}) = \texttt{true}$.

Listing 1 depicts the combination strategy. Valid test inputs are generated such that they satisfy all constraints, i.e. $\forall \tau : \Gamma(\tau, C^{ex} \cup C^{err}) = \texttt{true}$. Instead of iterating through all invalid values, strong invalid test inputs are generated by iterating through all error-constraints one at a time. Then, the currently selected error-constraint c_i is negated and test inputs are generated such that all constraints including $\overline{c_i}$ but excluding c_i are satisfied, i.e. $\forall \tau : \Gamma(\tau, C^{ex} \cup C^{err} \setminus \{c_i\} \cup \{\overline{c_i}\}) = \texttt{true}$.

IV. Related Work

Robustness is often tested by automated robustness testing tools [12]. The SUT is stimulated using random values and boundary values based on parameter types in order to crash the SUT. This approach does not use information from domain experts or specification. In comparison, the objective of our combinatorial robustness testing is to check whether business rules are implemented correctly. For instance, an invalid test input like `[Title:Mrs, GivenName:John, FamilyName:123]` is expected to yield an error but the implementation has a fault and does accept the invalid test input. This is usually not detected by automated robustness testing tools.

In contrast, black-box testing techniques are able to find missing but required functionality. CT tools like AETG [5], ACTS [11] and PICT [6] include the concept of invalid values. Invalid value combinations are not directly considered. To model them, a workaround is required [7]. In contrast, an approach we proposed in previous work [7] directly considers invalid values and invalid value combinations.

Base-choice is another coverage criteria and combination strategy that supports invalid values. Base-choice coverage subsumes single error coverage if the base test input is contains only valid values [3]. But, test inputs generated with base-choice do not satisfy single error t-wise coverage.

Grindal, Offut and Andler [3] survey combination strategies and discuss coverage criteria for invalid test inputs. Wojciak and Tzoref-Brill [13] report on system level combinatorial testing that includes testing of negative scenarios. In their case,

DOI 10.1109/QRS-C.2019.00045

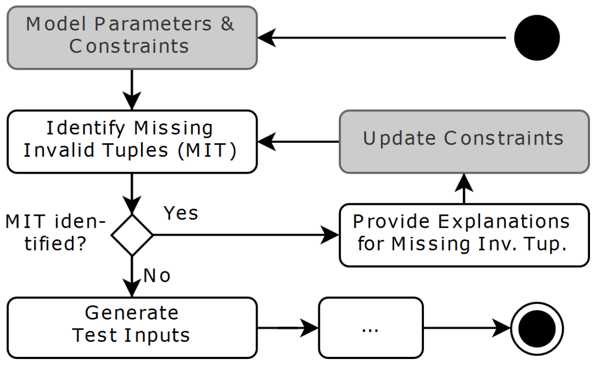
Model Parameters & Constraints
Identify Missing Invalid Tuples (MIT)
Update Constraints
MIT identified?
Yes
Provide Explanations for Missing Inv. Tup.
No
Generate Test Inputs
...

Figure 2. CT & Repair Process

single error coverage is not sufficient because error-handling depends on interactions between invalid and valid values. In a case study [14], we analyzed bug reports of a software for life insurances. As a conclusion, only considering invalid values is insufficient for applications with complex input domains.

Yilmaz et al. [4] reduce the input masking effect iterative by analyzing test results and generating new test inputs. The focus is on masking due to failing test cases and not due to error-handling. The objective of Gargantini, Petke and Radavelli [15] is also to identify and repair constraints which are too strong or too weak. Their approach is based on actual execution and they purposely generate test inputs that violate some constraints.

V. Repairing Over-Constrained Models for Combinatorial Robustness Testing

A. Overview

Compared to traditional CT, the robustness extensions have the advantage of avoiding potential input masking. However, it is easy to create over-constrained models when modeling invalid value combinations. In fact, real-world models are often over-constrained and both approaches can suffer from over-constrained models. When constraints are not correctly modelled, an invalid value or invalid value combination might not appear in test inputs and faults can remain undetected [7].

In this work, we focus on a special case of over-constrained models. Therefore, we assume that the parameters and values are correctly modelled and the constraints are conflict-free for valid test input generation. But, when generating invalid test inputs, conflicts between constraints arise and prevent invalid values or invalid value combinations from appearing. Then, a tester must repair the model by relaxing constraints.

Relaxation is often a labour-intensive and complicated task. Therefore, we propose a process to repair an over-constrained model that is supported by an automatic identification of missing invalid tuples and by an automatic identification of suspect constraints (explanation). The process is depicted in Figure 2. The grey background indicates manual activities while the other activities can be automated.

After modelling the parameters with values and constraints, the model is analyzed to identify invalid values and invalid value combinations that are missing, e.g. missing invalid tuples (MIT) that do not appear in the test suite because of contradicting constraints.

When no missing invalid tuples are identified, the *normal* CT process continues with test input generation and execution.

But, when missing invalid tuples are identified, the tester needs an explanation, e.g. a list of constraints which are suspected to be incorrect, that helps to understand why an invalid tuple is missing. Using the explanation, the tester can analyze and update the suspect constraints. The tester must decide to either remove the missing invalid tuple from the test or to relax the conflicting constraints. Afterwards, the model is again analyzed and either remaining MITs are identified or test input are generated.

In the following, over-constrained models are defined. Also, the activities of identifying missing invalid tuples and providing explanations are described in more detail.

The presented concepts are based on the second approach to combinatorial robustness testing. However, they can be transferred to the first approach.

B. Defining Over-Constrained Models

Reconsider the modified example with the rewritten constraints c_4' and c_5' that use the $=$ operator instead of $\neq$. The modifications are semantically equal for valid test input generation but affect the generation of invalid test inputs.

$$c_4' : Title = Mrs \Rightarrow GivenName = Jane$$
$$c_5' : Title = Mr \Rightarrow GivenName = John$$

To further explain the effect, we use another view on error-constraints. An error-constraint c_i is a specification of a set of invalid tuples (denoted as $\mathcal{I}_i$), i.e. a specification of either invalid values or invalid value combinations. While each invalid tuple $\tau \in \mathcal{I}_i$ must not appear in any valid test input, each invalid tuple is expected to appear in at least one or more strong invalid test inputs to satisfy single error coverage or single error t-wise coverage, respectively.

Obtaining the set of invalid tuples for an error-constraint means to reason about the conditions described by the single error-constraint. This is contrary to a constraint solver which reasons about a set of constraints and their inter-dependencies.

A single error-constraint c_i describes conditions for a non-empty set of k_i parameters. The parameter-indices for error-constraint c_i are denoted as $\phi_i = \{l_1^i, ..., l_{k_i}^i\}$. For instance, error-constraint c_3 of the example (Listing 1) constraints only values of `FamilyName` whereas error-constraint c_4 restricts value combinations of `Title` and `GivenName`. The sets of parameter-indices are $\phi_3 = \{3\}$ and $\phi_4 = \{1, 2\}$, respectively.

To calculate the set of invalid tuples $\mathcal{I}_i$, the cartesian product of values for the parameter-indices ϕ_i is computed and filtered such that each tuple does not satisfy the error-constraint c_i.

$$\mathcal{I}_i = \{\tau | \tau \in V_{l_1^i} \times ... \times V_{l_{k_i}^i} : \Gamma(\tau, \{c_i\}) = \texttt{false}\}$$

For instance, error-constraint c_5 describes conditions for the parameter-subset $\phi_5 = \{1, 2\}$. The cartesian product $V_1 \times V_2$

DOI 10.1109/QRS-C.2019.00045

is {`[Title:Mr, GivenName:John], ..., [Title:123, GivenName:123]`}. The subset which does not satisfy error-constraint c_5 denotes the set of invalid tuples $\mathcal{I}_5$. All invalid tuples of the example are shown below.

$\mathcal{I}_1 =$ {`[Title:123]`}
$\mathcal{I}_2 =$ {`[GivenName:123]`}
$\mathcal{I}_3 =$ {`[FamilyName:123]`}
$\mathcal{I}_4 =$ {`[Title:Mrs, GivenName:John]`}
$\mathcal{I}_5 =$ {`[Title: Mr, GivenName:Jane]`}

In contrast, the rewritten constraints c'_4 and c'_5 describe slightly different sets of invalid tuples:

$\mathcal{I}'_4 =$ {`[Title:Mrs, GivenName:John]`, `[Title:Mrs, GivenName:123]`}
$\mathcal{I}'_5 =$ {`[Title:Mr, GivenName:Jane]`, `[Title:Mr, GivenName:123]`}

Invalid tuples have a dual role in the generation process. When generating valid test inputs or invalid test inputs for an error-constraint other than c_i, the invalid tuples $\tau \in \mathcal{I}_i$ must not appear in any test input. When generating invalid test inputs for error-constraint c_i, each invalid tuple $\tau \in \mathcal{I}_i$ must appear in at least one test input. For instance, at least one invalid test input for c_1 must contain `[Title:123]`. Based on the other invalid tuples and their inter-dependencies, the test inputs for c_1 must not contain `[GivenName:123]`. A possible solution is `[Title:123, GivenName:John, FamilyName:Doe]`.

In the modified example, two problems can be observed. Because of error-constraint c_2, some invalid test inputs must contain $\mathcal{I}_2 =$ {`[GivenName:123]`}. However, due to $\mathcal{I}'_4$ and $\mathcal{I}'_5$, `[GivenName:123]` cannot be combined with `[Title:Mr]` and `[Title:Mrs]` as these are forbidden by $\mathcal{I}'_4$ and $\mathcal{I}'_5$. Due to $\mathcal{I}_1$, `[Title:123]` is not possible as well. $\mathcal{I}_1, \mathcal{I}'_4$ and $\mathcal{I}'_5$ implicitly forbid `[GivenName:123]`. Hence, there is a contradiction between error-constraint c_2 and c_1, c'_4, c'_5 which needs to be repaired. Two other contradictions are related to `[Title:Mrs, Given Name:123]` of $\mathcal{I}'_4$ and `[Title:Mr, GivenName:123]` of $\mathcal{I}'_5$. They are explicitly forbidden by error-constraint c_2 with $\mathcal{I}_2 =$ {`[GivenName:123]`}.

To further investigate the problem, conflicts and over-constrained models are defined.

Definition 1: A **conflict** is a contradiction between error-constraint c_i and some other constraints $C^{err} \backslash \{c_i\} \cup C^{ex}$. The interaction between c_i and some other constraints explicitly or implicitly prevents an invalid tuple $\tau \in \mathcal{I}_i$ as specified by c_i from being covered by at least one strong invalid test input.

$$\exists \tau \in \mathcal{I}_i \text{ such that } \Gamma(\tau, C^{err} \backslash \{c_i\} \cup C^{ex}) = \texttt{false}$$

Conversely, **no conflict** exists for error-constraint c_i if and only if for each invalid tuple $\tau \in \mathcal{I}_i$, a test input exists that covers the invalid tuple and satisfies all other constraints.

$$\forall \tau \in \mathcal{I}_i : \Gamma(\tau, C^{err} \backslash \{c_i\} \cup C^{ex}) = \texttt{true}$$

$p_1 : T$	$V_1 = \{1, 2, 3\}$
$p_2 : G$	$V_2 = \{1, 2, 3\}$
$p_3 : F$	$V_3 = \{1, 2\}$
$c_1 : T \neq 3$	
$c_2 : G \neq 3$	
$c_3 : F \neq 2$	
$c_4 : T = 2 \Rightarrow G \neq 1$	
$c_5 : T = 1 \Rightarrow G \neq 2$	

Figure 3. Internal Representation of Exemplary IPM

Definition 2: A model is **over-constrained** if and only if at least one conflict of an error-constraint exists. Otherwise, the model is not over-constrained.

C. Identifying Missing Invalid Tuples

To repair an over-constrained model, all conflicts must be removed by relaxing one or more constraints. First, all invalid tuples that are specified by an error-constraint but not covered in the final test suite must be identified. Then, the tester must decide for each missing invalid tuple if it should be removed from the test or if the constraints that cause the absence should be relaxed.

Definition 3: An invalid tuple $\tau \in \mathcal{I}_i$ specified by error-constraint c_i is called a **missing invalid tuple** if and only if other error-constraints $C^{err} \backslash \{c_i\}$ or exclusion-constraints prevent it from appearing in any test input. In other words, invalid tuple $\tau \in \mathcal{I}_i$ is missing because of a conflict for c_i that prevents τ from appearing in any test input.

$$\tau \in \mathcal{I}_i \text{ is missing iff } \Gamma(\tau, C^{err} \backslash \{c_i\} \cup C^{ex}) = \texttt{false}$$

When generating relevant test inputs, constraint handling is involved every time a tuple is created or changed. It is used to consider only those values and value combinations that satisfy the constraints. For that purpose, the IPM, constraints and tuple under change are transformed into a constraint satisfaction problem (CSP) [10]. A CSP consists of three components X, D and C [16]. X is a set of variables, D is a set of domains with one domain for each variable and C is a set of constraints that restricts value combinations of variables. A solution for a CSP is an assignment of values to variables which is both consistent and complete. An assignment that does not violate any constraint is consistent. Otherwise, the assignment is inconsistent. An assignment is complete if every variable has a value assigned. Otherwise, it is partial. A SAT-solver is applied to find a solution for the CSP. If a solution is found, the tuple is accepted and can be further used in test input generation. If no solution exists, the tuple cannot be used since one or more constraints cannot be satisfied.

For the sake of clarity, Figure 3 depicts the internal representation of our example. Based on that internal representation, the translation into a CSP is done as follows. Each parameter p_i of the IPM is represented as a variable $x_i \in X$. The domain of x_i represents the values V_i of parameter p_i as integers $D_{x_i} = \{1, ..., m_i\}$. All constraints are translated to constraints of the CSP. The parameter `Title` is represented as the variable `T` and its values are $D_T = \{1, 2, 3\}$. Variable `G` represents `GivenName` and `F` represents `FamilyName`. Constraints are translated accordingly. For instance, $Title \neq 123$ becomes

DOI 10.1109/QRS-C.2019.00045

$T \neq 3$. The values of the tuple are also added as constraints. We refer to them as tuple-constraints. A tuple `[Title:Mr, GivenName:John]` translates to $\{T = 1, G = 1\}_\tau$.

$$\begin{aligned} &X = \{T, G, F\} \\ &D = \{D_T = \{1,2,3\}, D_G = \{1,2,3\}, D_F = \{1,2\}\} \\ &C = \{T \neq 3, G \neq 3, F \neq 2, T = 2 \Rightarrow G \neq 1, T = 1 \Rightarrow G \neq 2\} \\ &\qquad \cup \{T = 1, G = 1\}_\tau \end{aligned}$$

To identify missing invalid tuples for an error-constraint c_i with one or more conflicts, another CSP can be used prior to the generation of invalid test inputs. For each invalid tuple $\tau \in \mathcal{I}_i$, it is checked if at least one test input exists that covers the invalid tuple and satisfies all other constraints. Therefore, the above shown CSP is modified for each invalid tuple $\tau_j \in \mathcal{I}_i$. The invalid tuple is modelled as a tuple-constraint $\{...\}_{\tau_j}$ and the specifying error-constraint c_i is removed.

The set of all missing invalid tuples for error-constraint c_i is denoted as $\mathcal{M}_i$.

$$\mathcal{M}_i = \{\tau | \tau \in \mathcal{I}_i : \Gamma(\tau, C^{err} \backslash \{c_i\} \cup C^{ex}) = \texttt{false}\}$$

D. Explaining Conflicts

Knowing the missing invalid tuples is a necessary first step to repair an over-constrained model. Understanding the absence of a missing invalid tuple is the next step. Then, the tester must either discard the missing invalid tuple from testing by relaxing the error-constraint that is contradictory (c_i) or the tester must relax other related constraints $C^{err} \backslash \{c_i\} \cup C^{ex}$.

To further explain the absence of a missing invalid tuple caused by a conflict, we introduce the notion of conflict sets.
Definition 4: A **conflict set** $\mathcal{O} \subseteq C^{err} \backslash \{c_i\} \cup C^{ex}$ is a set of constraints that explains the absence of a missing invalid tuple. In other words, no invalid test input exists that covers τ while satisfying the constraints of the conflict set.

$$\forall \tau \in \mathcal{M}_i \; \exists \mathcal{O} : \Gamma(\tau, \mathcal{O}) = \texttt{false}$$

Technically, a conflict a for missing invalid tuple $\tau \in \mathcal{M}_i$ can be explained by the set of all other constraints ($\mathcal{O} = C^{err} \backslash \{c_i\} \cup C^{ex}$). But, since repairing is a manual labour-intensive task, dealing with many constraints is often not useful because conflicts can become unclear and confusing. In addition, only a subset of constraints is responsible for the conflict and not every relaxation of constraints resolves the conflict. Rather than dealing with all constraints, it is more useful to identify and deal with a smaller subset of constraints that explains the conflict and can be repaired. Therefore, we search for a minimal conflict set as an explanation that consists of as few constraints as possible.
Definition 5: A conflict set $\mathcal{O}$ for a missing invalid tuple $\tau \in \mathcal{M}_i$ is a **minimal conflict set** if and only if there exists no proper subset $\mathcal{O}' \subset \mathcal{O}$ that can explain the conflict as well.

$$\mathcal{O} \text{ is minimal iff } \nexists \mathcal{O}' \subset \mathcal{O} \text{ for which } \Gamma(\tau, \mathcal{O}') = \texttt{false} \text{ holds}$$

For instance, consider invalid test input generation for error-constraint c'_4 of the example. To have an explanation for the one missing invalid tuple `[Title:Mrs, GivenName:123]`, the minimal conflict set $\mathcal{O} = \{c_2\}$ can be identified. Without searching, the conflict set containing all other error-constraints $\mathcal{O} = \{c_1, c_2, c_3, c'_5\}$ would function as an explanation. Because of the small example, the difference between the two conflict sets is small. However, the benefit of minimal conflict sets increases for real-world models with many more constraints.

Constraints and conflicts are interconnected, e.g. a constraint can belong to several conflict sets and a relaxation of one constraint can solve several conflicts. Therefore, the iterativ process is reasonable where the tester focuses on one missing invalid tuple with one minimal conflict set at a a time. Afterwards, the updated model is re-analyzed to identify invalid tuples that are still missing. To further reduce the work, a tester can mark already analyzed or relaxed constraints as being *sound*. Sound constraints are considered correct and are not again proposed as an explanation. Thereby, the tester can reduce the size of minimal conflict sets and avoid double work.

The set of sound constraints (denoted as $\mathcal{S}$) is a subset of all relaxable constraints, e.g. $\mathcal{S} \subset C^{err} \backslash \{c_i\} \cup C^{ex}$.

To identify minimal conflict sets, we rely on conflict detection techniques for CSP. Generally speaking, if there exists no solution for a CSP, some constraints must be relaxed to restore consistency [17]. Therefore, constraints are partitioned into two disjoint groups: Constraints that can be relaxed (denoted as $\mathcal{C}$) and so-called *background* constraints that cannot be relaxed (denoted as $\mathcal{B}$) are distinguished. If no solution exists for the constraints $\mathcal{C} \cup \mathcal{B}$, the model is over-constrained. A proper subset $\mathcal{R} \subset \mathcal{C}$ is a relaxation if and only if a solution exists for $\mathcal{R} \cup \mathcal{B}$. However, no relaxation exists if $\mathcal{B}$ is inconsistent. Again, a conflict set $\mathcal{O}$ denotes a subset of constraints $\mathcal{O} \subseteq \mathcal{C}$ if and only if no solution exists for $\mathcal{O} \cup \mathcal{B}$.

To find a minimal conflict, an algorithm like `QuickXPlain` [17] can be used. Briefly explained, it is a divide and conquer approach that inspects constraints and determines whether they belong to the conflict or to the relaxation. To utilize `QuickXPlain`, it is necessary to identify the background constraints $\mathcal{B}$. Since the conflict set should explain why an invalid tuple $\tau \in \mathcal{M}_i$ is missing, $\overline{c_i}$ and the tuple-constraints should not be relaxed. All constraints marked as sound should also not be relaxed: $\mathcal{B} = \{\overline{c_i}\} \cup \{...\}_\tau \cup \mathcal{S}$. In contrast, the remaining unsound exclusion- and error-constraints are potentially too strict and should be relaxed: $\mathcal{C} = C^{err} \backslash \{c_i\} \cup C^{ex} \backslash \mathcal{S}$.

For the example, the constraints are separated as shown below. The last subset is empty because the example has no exclusion-constraints. The minimal conflict set is $\mathcal{O} = \{c_2\}$.

$$\begin{aligned} \mathcal{M}_4 &= \{\texttt{[Title:Mrs, GivenName:123]}\} \\ \mathcal{B} &= \{\overline{c'_4}\} \cup \{T = 2, G = 3\}_\tau \\ \mathcal{C} &= \{c_1, c_2, c_3, c'_5\} \cup \{\} \end{aligned}$$

Once missing invalid tuples and minimal conflict sets are identified, the model must be updated. Either the negated error-constraint $\overline{c_i}$ or the conflicting constraints $\mathcal{O}$ must be manually relaxed until no more invalid tuples are missing. Afterwards, invalid test inputs can be generated such that they satisfy single error coverage or single error t-wise coverage.

Table I
TEST MODELS USED FOR EXPERIMENTS

Name	**IPM**	**C^{err}**	**Invalid**	**Missing**
Example-1	$3^2 2^1$	$1^3 2^2$	5	0
Example-2	$3^2 2^1$	$1^3 2^2$	7	3
Registration	$6^1 4^4 3^1 2^9$	$2^{11} 1^6$	41	11
Banking-1	$4^1 3^4$	5^1	112	0
Ba-1 +10	$4^1 3^4$	5^2	122	20
Ba-1 +50	$4^1 3^4$	5^2	162	100
Banking-2	$4^1 2^1 4$	2^1	3	0
Ba-2 +1	$4^1 2^1 4$	2^2	4	2
Ba-2 +1 +1	$4^1 2^1 4$	2^3	5	4
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{10}	31	0
HC-3 +3 +4 +5	$6^1 5^1 4^5 3^6 2^{16}$	2^{13}	43	24
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^7	22	0
HC-4 +3 +4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^9	29	7

VI. EXPERIMENTS

A. Experimental Setup

We developed a prototype of the robustness extension including the detection and explanation of conflicts. It is integrated into our CT framework [18]. The source code and experiments are available at our companion website[1].

The objective of the experiments is to evaluate the feasibility of the proposed conflict detection and explanation technique. Therefore, we generate test inputs and compare the test suite sizes and times required for test input generation with enabled conflict explanation versus disabled conflict explanation.

Table I lists the used benchmark IPMs. `Example1` and `Example2` are the unmodified and modified examples used throughout this paper. `Registration` is a real-world IPM from one of our industry cooperation partners. The other IPMs originate from [19] and are often used to compare combination strategies. For these experiments, they are modified such that all constraints are error-constraints.

The identifier `A` $+n$ indicates that IPM `A` is extended by an error-constraint that specifies n missing invalid tuples. For instance, two error-constraints are added to `Ba-2 +1 +1`. Each one specifies one invalid value that is already specified for `Banking 2`. The `IPM` describes the parameter values in exponential notation; x^y refers to y parameters with x values. $\mathtt{C}^{\mathtt{Err}}$ uses exponential notation to describe error-constraints. Here, x^y refers to y error-constraints for x parameters. The `Invalid` column states the number of invalid tuples specified by all error-constraints and `Missing` states the subset of missing invalid tuples for which no test input is generated.

The experiments are carried out as a build job on a server with an Intel Xeon 1.9 GHz CPU and 16 GB of memory. The experiments are repeated for the strengths $t = 1$ and $t = 2$ since these are mostly used.

B. Data and Analysis

The results of the experiments are summarized in Table II. The `t` column refers to the testing strength, `Size` refers to the sizes of test suites. The columns `Disabled` and `Enabled` list the measured times in seconds for test input generation with conflict explanation disabled and enabled, respectively.

[1]https://github.com/coffee4j/cta-2019

Table II
RESULTS OF EXPERIMENTS

Name	**t**	**Size**	**Disabled**	**Enabled**	**Diff.**
Example-1	1	10	0.003 s	0.004 s	0.001 s
	2	10	0.004 s	0.004 s	0.000 s
Example-2	1	8	0.005 s	0.008 s	0.003 s
	2	8	0.005 s	0.008 s	0.003 s
Registration	1	109	1.556 s	1.691 s	0.135 s
	2	372	17.028 s	17.195 s	0.167 s
Banking-1	1	116	0.552 s	0.812 s	0.260 s
	2	128	1.058 s	1.325 s	0.266 s
Ba-1 +10	1	106	1.072 s	5.095 s	4.022 s
	2	118	1.649 s	5.636 s	3.987 s
Ba-1 +50	1	66	1.475 s	31.671 s	30.196 s
	2	78	2.280 s	32.392 s	30.112 s
Banking-2	1	11	0.019 s	0.019 s	0.000 s
	2	51	0.196 s	0.194 s	-0.002 s
Ba-2 +1	1	9	0.026 s	0.028 s	0.002 s
	2	39	0.274 s	0.278 s	0.004 s
Ba-2 +1 +1	1	7	0.033 s	0.038 s	0.004 s
	2	27	0.371 s	0.374 s	0.003 s
HealthCare-3	1	230	4.610 s	4.624 s	0.014 s
	2	1431	109.657 s	109.699 s	0.041 s
HC-3 3 +3 +4 +5	1	145	5.220 s	5.839 s	0.619 s
	2	849	135.984 s	136.027 s	0.042 s
HealthCare-4	1	161	3.558 s	3.602 s	0.044 s
	2	1280	123.324 s	123.268 s	-0.056 s
HC-4 +3 +4	1	112	3.860 s	4.064 s	0.204 s
	2	882	147.304 s	147.596 s	0.292 s

The experiments reveal a small overhead in generation time caused by the additional constraint solving for identification and explanation of minimal conflict sets. On average, the measured overhead, i.e. the absolute difference between generation times with and without explanation, is 2.706 seconds. For experiments with correct models without any conflicts, the average overhead is only 0.056 seconds. In contrast, the average of all experiments with over-constrained models is 4.362 seconds. Two experiments (`Ba-1 +10` and `Ba-1 +50`) are significantly slower than the other experiments with the highest measured overhead of 30.196 seconds. They are slower because more missing invalid tuples (10 and 50) must be identified and explained. In addition, the constraints involve five parameters whereas all other scenarios only have constraints that involve two parameters. Therefore, more searches are necessary and the search space is much larger.

In relative terms, enabling identification and explanation increases the generation time by 20.70% on average. The two slow experiments have a relative overhead ranging from 70.73% up to 95.34%. Ignoring the slower experiments results in an overhead of only 9.11% for the remaining experiments.

In two cases, the time measured for the generation with explanation even decreases slightly by 0.056 seconds. Differences in the range of milliseconds can be interpreted as environmental noise because the execution cannot be completely controlled. However, the small differences highlight that the additional overhead can be neglected.

The experiments also show that the identification and explanation are independent from the specified testing strength.

Increasing the testing strength from one to two increases the generation time on average by 39.655 seconds. In contrast, practically no difference for identification and explanation could be measured. On average, the difference is -0.0049 seconds which can be explained by noise.

In summary, the experiments indicate that the additional computations for identification of missing invalid tuples and explanation with minimal conflict sets is feasible and applicable to real world scenarios. In most cases, the measured overhead is very little. But even for the significant overhead, we believe that the manual tasks benefit from knowing why invalid tuples are missing and the additional time is worth it.

VII. Threads to Validity

We compared test input generation with conflict explanation enabled versus conflict explanation disabled. Since the comparison is based on actual test input generation, our results might depend on the implementation of the algorithms and on the design of the test scenarios. To ensure an unbiased implementation, we follow the suggestions for an efficient implementation by Kleine and Simos [20]. The test scenarios are based on existing benchmark IPMs. The implementation and test scenarios are published to allow a replication of the experiments[1]. During execution of experiments, resource consumption of other applications may have distorted the results. Therefore, the measurements are based on 30 repetitions.

VIII. Conclusion & Future Work

Combinatorial robustness testing is an extension for CT to test negative scenarios. They generate separate sets of test inputs to avoid the input masking effect. Therefore, they require additional semantic information to separate valid from invalid test inputs. While the approaches work in general, it is easy to create over-constrained models. Then, not all specified invalid values and invalid value combinations appear in the test inputs and faults could remain undetected.

In this paper, over-constrained models and conflicts are defined to further explain the phenomenon of missing invalid tuples. To support the labour-intensive manual work of repairing over-constrained models, the necessary steps are integrated into a CT & repair process. In addition, automated techniques are presented to identify missing invalid tuples and to provide explanations in form of minimal sets of conflicting constraints.

The presented techniques are implemented in a Java-based prototype called `coffee4j` and experiments are conducted using a set of benchmark models. The source code and experiments are available at our companion website[1].

The experiments indicate that the overhead for the identification and explanation is rather small and negligible. Enabling the techniques leads to no relevant overhead for not over-constrained models but to some overhead for over-constrained models. Although, we believe the benefits of the better explanation in manual work are worth the additional time in automated generation work.

In future work, we will focus on techniques that allow to generate invalid test inputs directly from over-constrained models and on techniques that allow to repair over-constrained models automatically.

References

[1] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610.12-1990, 1990.

[2] P. Sawadpong, E. B. Allen, and B. J. Williams, "Exception handling defects: An empirical study," in *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, Oct 2012, pp. 90–97.

[3] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, 2005.

[4] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. Porter, "Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, 2014.

[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*, vol. 23, no. 7, 1997.

[6] J. Czerwonka, "Pairwise testing in real world," in *24th Pacific Northwest Software Quality Conference*, 2006.

[7] K. Fögen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *2018 IEEE Eleventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 7th International Workshop on Combinatorial Testing (IWCT)*, 2018.

[8] ——, "Combinatorial robustness testing with negative test cases," in *2019 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2019.

[9] M. Grindal, J. Offutt, and J. Mellin, "Managing conflicts when using combination strategies to test software," in *2007 Australian Software Engineering Conference (ASWEC'07)*, April 2007, pp. 255–264.

[10] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013.

[11] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 370–375.

[12] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology*, vol. 55, no. 1, 2013.

[13] P. Wojciak and R. Tzoref-Brill, "System level combinatorial testing in practice - The concurrent maintenance case study," *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, 2014.

[14] K. Fögen and H. Lichter, "A case study on robustness fault characteristics for combinatorial testing - results and challenges," in *2018 6th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2018) co-located with APSEC 2018*, 2018.

[15] A. Gargantini, J. Petke, and M. Radavelli, "Combinatorial interaction testing for automated constraint repair," in *2017 IEEE Tenth International Conference on Software Testing, Verification and Validation Workshops, ICSTW, 6th International Workshop on Combinatorial Testing (IWCT)*, 2017.

[16] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*. Prentice Hall PTR, 2010.

[17] U. Junker, "QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems," in *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, 2004.

[18] J. Bonn, K. Fögen, and H. Lichter, "A framework for automated combinatorial test generation, execution, and fault characterization," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2019, pp. 224–233.

[19] 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*, ser. ISSTA '11. ACM, 2011.

[20] K. Kleine and D. E. Simos, "An efficient design and implementation of the in-parameter-order algorithm," *Mathematics in Computer Science*, vol. 12, no. 1, 2018.