Semi-Automatic Repair of 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*—Combinatorial robustness testing is an approach to generate separate test inputs for positive and negative test scenarios. The test model is enriched with semantic information to distinguish valid from invalid values and value combinations. Unfortunately, it is easy to create over-constrained models and invalid values or invalid value combinations do not appear in the final test suite. In this paper, we extend previous work on manual repair and develop a technique to semi-automatically repair over-constrained models. The technique is evaluated with benchmark models and the results indicate a small computational overhead.**

***Keywords*-Robustness Testing, Combinatorial Testing**

I. Introduction

Testing with invalid inputs is important to check the robustness property of software systems. Robustness describes “the degree to which a system or component can function correctly” in the presence of invalid inputs [1]. Invalid inputs are inputs to the system that contain 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. Often, error-handlers are implemented to make a system robust by appropriately reacting to external faults. Unfortunately, they can contain up to three times more faults than normal source code [2].

Invalid values and invalid value combinations can cause input masking [3]. Once the SUT starts evaluating invalid input, the SUT detects the external fault, initiates error-handling and responds with an error message. Then, the remaining values and value combinations of the test input are not tested.

Therefore, combinatorial robustness testing (CRT) is an extension to combinatorial testing (CT) that generates separate test suites of valid and invalid test inputs [3]. Similar to CT, parameters, values and exclusion-constraints to exclude irrelevant value combinations. For CRT, the test model also contains additional semantic information to mark certain values and value combinations as invalid.

The generation of valid and invalid test inputs is separated. First, invalid values and invalid value combinations are excluded from valid test input generation. Afterwards, invalid test inputs are generated iteratively and a loop is used to traverse all invalid values and invalid value combinations one at a time.

Two different approaches exist. The first one allows to mark single values as invalid by separating valid from invalid values [4], [5], [6]. But, invalid value combinations cannot be modelled directly. The second approach uses a second set of constraints (error-constraints) to directly describe invalid values as well as invalid value combinations [3].

Both approaches have in common that it is easy to create over-constrained models when applying CRT in practice [7]. As a consequence, not all specified invalid values and invalid value combinations appear in test inputs (missing invalid tuples) and faults could remain undetected.

In previous work, we discussed techniques to explain over-constrained models by identifying missing invalid tuples conflicting constraints [7]. Based on the explanations, the tester can manually repair the model by relaxing identified constraints to remove one or more conflicts. However, manual repair of constraints could be rejected since it can be perceived as too time-consuming, too costly or too complex. Instead, an automation of model repair activities would be desirable.

In this paper, we extend our previous work on manual repair of over-constrained models and present a technique for automatic and semi-automatic repair. Furthermore, we argue why the semi-automatic application is preferable.

The paper is structured as follows. First, an example is presented to illustrate over-constrained models and missing invalid tuples. Section III and Section V summarize foundations of CRT and related work. The concept for semi-automatic repair of over-constrained models is discussed in Section IV. In Section VI, we provide an evaluation. Afterwards, we conclude with a summary of our work.

II. Example

Throughout the paper, we reuse an example from previous work [7]. It is a simple customer registration service with validity checks to ensure that the entered data matches the intended semantics of the input fields. The following checks have to be done: Empty inputs should be avoided and a person’s title should match the gender of the given name. Since the service cannot correct wrong data itself, it should return an error message asking the user to correct the data.

A test model for the registration service is depicted in Fig.1. Error-constraints describe invalid values like `[GivenName: 123]` and invalid value combinations like `[Title:Mrs, GivenName:John]`. For instance, test input `[Title:Mr, Gi-`

To be published in proceedings of APSEC 2019, Putrajaya, Malaysia

$$\begin{array}{ll} p_1 : Title & V_1 = \{Mr, Mrs, 123\} \\ p_2 : GivenName & V_2 = \{John, Jane, 123\} \\ p_3 : FamilyName & V_3 = \{Doe, Foo, 123\} \\ \hline c_1 : Title \neq 123 & \\ c_2 : GivenName \neq 123 & \\ c_3 : FamilyName \neq 123 & \\ c_4 : Title = Mrs \Rightarrow GivenName = Jane & \\ c_5 : Title = Mr \Rightarrow GivenName = John & \end{array}$$

Fig. 1. Exemplary Test Model with Five Error-Constraints

`venName:John, FamilyName:Doe]` is valid because it satisfies all constraints. In contrast, test input `[Title:Mrs, GivenName:Jane, FamilyName:123]` is invalid because it contains a family name that does not satisfy error-constraint c_3. It is also an example of input masking. Because of the family name, an error message is returned 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 adhere to a given combinatorial coverage criterion [3]. 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 input of which all other values and value combinations are valid. For the given example, at least one test input should contain `[Title:123]` as specified by error-constraint c_1 which satisfies all other error-constraints.

The depicted test model is model over-constrained and not all specified invalid values or invalid value combinations can appear in test inputs. As a result, faults can remain undetected.

CRT finds no invalid test input for `[GivenName:123]` of error-constraint c_2 that satisfies all other constraints: The combinations `[Title:Mr, GivenName:123]` and `[Title: Mrs, GivenName:123]` do not satisfy error-constraints c_4 and c_5. `[Title:123, GivenName:123]` does not satisfy error-constraint c_1. To remove this conflict, either c_1, c_4 or c_5 must be relaxed. For instance, c_4 could be slightly rewritten with the $\neq$ operator instead of $=$ as depicted by c_4'.

$$c_4' : \ Title = Mrs \Rightarrow GivenName \neq John \qquad (1)$$

III. Foundations of Combinatorial Robustness Testing

In this section, foundations of CRT and techniques to identify, explain and manually repair over-constrained models are briefly recapitulated. The concepts and explanations are based on the second approach with error-constraints. But, they can be transferred to the first approach as well. A more detailed discussion can be found in previous work [3], [7].

A. Basic Definitions

CRT is a black-box testing approach that systematically generates test inputs based on a given test model. It is an extension of CT and separates the generation of valid and invalid test inputs [3]. The **test model** is a quadruple $\mathcal{T} = \langle P, V, C^{ex}, C^{err} \rangle$ where P represents n input parameters $P = \{p_1, ..., p_n\}$. $V = \{V_1, ..., V_n\}$ represents all value sets where each non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$ is associated with an input parameter p_i. C^{ex} and C^{err} denote sets of constraints which are explained in subsequent paragraphs.

Let (p_i, v_j) denote a parameter-value pair such that value $v_j \in V_i$ is *assigned* to parameter p_i. A **tuple** τ is a set of parameter-value pairs for d distinct parameters such as `[Title:Mr, GivenName:John]`. A tuple with n parameter-value pairs 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 τ_a includes all parameter-value pairs of τ_b.

Real-world systems often have restrictions in their input domains and certain combinations of parameter values should not be combined [8]. These value combinations are **irrelevant** as they are, for instance, not executable or just not of any interest for the test. Therefore, irrelevant value combinations should be detected and removed from the test suite.

Constraint handling can be used to exclude unwanted values and value combinations [8]. Constraints are explicitly modeled as logical expressions that describe conditions [9]. A function $\Gamma(\tau, C) \rightarrow \texttt{Bool}$ evaluates if a tuple τ satisfies a set of constraints C. A constraint-aware generation algorithm generates test inputs such that all parameter values appear in the desired frequency while satisfying the constraints in C. Formally, $C^{ex} \in \mathcal{T}$ is a set of constraints to distinguish between relevant and irrelevant tuples which we denote **exclusion-constraints**. 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: $\Gamma(\tau, C^{ex}) = \texttt{false}$.

Relevant tuples can be further partitioned into valid and invalid tuples. Therefore, a separate set of constraints was introduced to describe invalid values and invalid value combinations: **error-constraints** (denoted as $C^{err} \in \mathcal{T}$). **Valid** tuples are relevant and do not contain any invalid value or invalid value combinations to prevent error-handling. Formally, a relevant tuple is **valid** if all exclusion-constraints are satisfied and if all error-constraints are satisfied as well: $\Gamma(\tau, C^{ex} \cup C^{err}) = \texttt{true}$. **Invalid** tuples are also relevant but contain at least one invalid value or one invalid value combination to trigger error-handling. While all exclusion-constraints are satisfied, at least one error-constraint remains unsatisfied. 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. Formally, exactly one error-constraint remains unsatisfied: $\exists! c \in C^{err} : \Gamma(\tau, \{c\}) = \texttt{false}$ and $\Gamma(\tau, C^{ex} \cup C^{err} \backslash \{c\}) = \texttt{true}$.

B. Generation of Strong Invalid Test Inputs

Valid test inputs are generated to satisfy a combinatorial coverage criterion like **t-wise coverage** (t denotes the testing strength) while excluding all values and value combinations that are irrelevant or invalid. It is satisfied if each valid value combination of t parameters appears in at least one valid test input [10]. Invalid test inputs are generated adhering to another coverage criterion. For instance, the **single error coverage criterion** is satisfied if each specified invalid value and each invalid value combination appears in at least one test input of which all other values are valid [10], [3].

Therefore, let `gen(P, V, C, t)` be a combination strategy that generates a set of test inputs for given input parameters `P`, their values `V` and testing strength `t` that satisfy a set of constraints `C`. For a given test model $\mathcal{T}$, valid test inputs are generated such that they satisfy all constraints, i.e. `gen(P, V,` $C^{ex} \cup C^{err}$`, t)`. Strong invalid test inputs are generated with a strength of $t = 0$ by iterating through all error-constraints one at a time. The currently selected error-constraint c_i is negated (denoted as $\overline{c_i}$) and a set of test inputs is generated that satisfies all constraints including $\overline{c_i}$ but excluding c_i, i.e. $\forall c_i \in C^{err}$, `gen(P, V,` $C^{ex} \cup C^{err} \backslash \{c_i\} \cup \{\overline{c_i}\}$`, t)`.

Another view on error-constraints is helpful for the following concepts: They have a dual role in the generation process. When generating valid test inputs, all error-constraints specify tuples that should be excluded. When generating invalid test inputs for error-constraint $\overline{c_i}$, all other error-constraints specify tuples that should be excluded. When generating invalid test inputs for error-constraint $\overline{c_i}$, the error-constraint specifies a set of invalid tuples, i.e. invalid values and invalid value combinations. We denote the set of invalid tuples specified by error-constraint $\overline{c_i}$ as $\mathcal{I}_i$. For instance, the following invalid tuples are specified by the example.

$$
\begin{aligned}
\mathcal{I}_1 &= \{\texttt{[Title:123]}\} \\
\mathcal{I}_2 &= \{\texttt{[GivenName:123]}\} \\
\mathcal{I}_3 &= \{\texttt{[FamilyName:123]}\} \\
\mathcal{I}_4 &= \{\texttt{[Title:Mrs, GivenName:John]}, \\
&\quad\ \texttt{[Title:Mrs, GivenName:123]}\} \\
\mathcal{I}_5 &= \{\texttt{[Title:Mr, GivenName:Jane]}, \\
&\quad\ \texttt{[Title:Mr, GivenName:123]}\}
\end{aligned} \tag{2}
$$

Each invalid tuple $\tau \in \mathcal{I}_i$ must not appear in valid test inputs. But, each invalid tuple is expected to appear in one or more strong invalid test inputs to satisfy single error coverage.

C. Identification & Explanation of Over-Constrained Models

Unfortunately, strong invalid test inputs cannot be generated if constraints are not correctly modelled. For further explanations, over-constrained models, conflicts and missing invalid tuples are defined.

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

Definition 2: A test model is **over-constrained** if and only if at least one conflict of an error-constraint exists.

Definition 3: An invalid tuple $\tau \in \mathcal{I}_i$ specified by error-constraint $\overline{c_i}$ is a **missing invalid tuple** if and only if a conflict with other error-constraints $C^{err} \backslash \{c_i\}$ or exclusion-constraints C^{ex} prevents it from appearing in any invalid test input.

The set of all missing invalid tuples for error-constraint $\overline{c_i}$ is denoted as $\mathcal{M}_i$. It is computed by checking for each invalid tuple, if at least one strong invalid test input can be generated.

$$
\mathcal{M}_i = \{\tau | \tau \in \mathcal{I}_i : \Gamma(\tau, C^{err} \backslash \{c_i\} \cup C^{ex}) = \texttt{false}\} \tag{3}
$$

For the example, the missing invalid tuples are as follows.

$$
\begin{aligned}
\mathcal{M}_2 &= \{\texttt{[GivenName:123]}\} \\
\mathcal{M}_4 &= \{\texttt{[Title:Mrs, GivenName:123]}\} \\
\mathcal{M}_5 &= \{\texttt{[Title:Mr, GivenName:123]}\}
\end{aligned} \tag{4}
$$

To explain the absence of a missing invalid tuple, we further introduce the notion of conflict sets.

Definition 4: A **conflict set** $\mathcal{O}_{i,j} \subseteq C^{err} \backslash \{c_i\} \cup C^{ex}$ is a set of constraints that explains the absence of a missing invalid tuple $\tau_j \in \mathcal{M}_i$. No invalid test input exists that covers τ_j while satisfying all constraints of the conflict set.

To repair an over-constrained model, all conflicts must be resolved by relaxing some constraints of the conflict sets. Oftentimes, only a subset of constraints is responsible for a conflict and not every relaxation of constraints resolves a conflict. Since repairing is a manual labour-intensive task, dealing with many constraints is often not useful because conflicts can become unclear and confusing. Instead, 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 $\mathcal{O}_{i,j}$ is **minimal** if and only if there exists no proper subset $\mathcal{O}'_{i,j} \subset \mathcal{O}_{i,j}$ to explain the conflict.

For the running example, the minimal conflict set for the missing invalid tuple $\tau_1 \in \mathcal{M}_4$ (`[Title: Mrs, GivenName: 123]`) is $\mathcal{O}_{4,1} = \{c_2\}$. In contrast, $\mathcal{O}_{4,1} = \{c_1, c_2, c_3, c_5\}$ is another conflict set which is not minimal and less helpful.

To support the tester in repairing the model, we proposed a repair process [7]. First, missing invalid tuples and one minimal conflict set for each missing invalid tuple are automatically identified. Then, the tester manually relaxes constraints as explained by a minimal conflict set. Automatic identification and manual relaxation repeat until all conflicts are removed and the model is repaired.

IV. Semi-automatic Repair of Over-Constrained Models

Using the previously described concepts, minimal conflict set $\mathcal{O}_{2,1} = \{c_1, c_4, c_5\}$ explains the absence of the missing invalid tuple $\tau_1 \in \mathcal{M}_2$ (`[GivenName:123]`). Due to the minimality property, only one of the three detected constraints must be relaxed to remove the conflict. However, it remains unclear (1) *which* constraint to select for relaxation and (2) *how* the selected constraint must be relaxed. The first aspect is discussed in the Subsection A and the second aspect is discussed in Subsection B. Afterwards, we argue why semi-automatic repair is preferable.

A. Automatic Diagnosis of Over-Constrained Models

In previous work [7], we used conflict detection algorithms like `QuickXplain` [11] to find a minimal conflict set $\mathcal{O}_{i,j}$ for a given missing invalid tuple $\tau_j \in \mathcal{M}_i$. To find a meaningful

To be published in proceedings of APSEC 2019, Putrajaya, Malaysia

conflict set, the constraints are partitioned into two disjoint groups [11]: Constraints that are relaxable (denoted as $\mathcal{C}$) and *background* constraints that cannot be relaxed (denoted as $\mathcal{B}$). 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. A subset of constraints $\mathcal{O} \subseteq \mathcal{C}$ denotes a conflict set $\mathcal{O}$ if and only if no solution exists for $\mathcal{O} \cup \mathcal{B}$ while $\mathcal{B}$ is consistent. Since the conflict set should explain why an invalid tuple $\tau \in \mathcal{M}_i$ is missing, $\overline{c_i}$ should not be relaxed. In contrast, the remaining exclusion- and error-constraints are potentially too strict and should be relaxed: $\mathcal{C} = C^{err} \backslash \{c_i\} \cup C^{ex}$.

While a minimal conflict set helps to identify subsets of constraints that explain a conflict, a tester must still manually decide which constraint of the subset to relax in order to solve the conflict. Since the absence of a missing invalid tuple can be caused by more than one conflict, it may be necessary to relax more than one constraint.

To determine all constraints that must be relaxed, a so-called diagnosis set can be computed [12]: In constraint handling, a diagnosis set is a set of constraints such that a model is repaired if all constraints of the diagnosis set are relaxed. If $\mathcal{B}$ is consistent and $\mathcal{C} \cup \mathcal{B}$ has no solution, a diagnosis set $\triangle \subseteq \mathcal{C}$ is a set of constraints such that $\mathcal{B} \cup \mathcal{C} - \triangle$ is consistent.

Definition 6: For a missing invalid tuple $\tau_j \in \mathcal{M}_i$, a **diagnosis set** $\triangle_{i,j} \subseteq C^{err} \backslash \{c_i\} \cup C^{ex}$ is a set of constraints such that all conflicts between error-constraint $\overline{c_i}$ and some other constraints can be removed by relaxing all constraints $c \in \triangle_{i,j}$. Formally, $\triangle_{i,j}$ is a diagnosis set if and only if $\Gamma(\tau_j, (C^{err} \backslash \{c_i\} \cup C^{ex}) - \triangle_{i,j}) = \texttt{true}$.

Technically, all constraints ($\triangle_{i,j} = C^{err} \backslash \{c_i\} \cup C^{ex}$) form a diagnosis set. However, it is preferable to relax a smaller subset of constraints instead. Therefore, we introduce the notion of minimal diagnosis sets [12].

Definition 7: A diagnosis set $\triangle_{i,j}$ is **minimal** if and only if no proper subset $\triangle'_{i,j} \subset \triangle_{i,j}$ is a diagnosis set.

Different algorithms to find minimal diagnosis sets exist [12]. A common approach is to compute a hitting set tree (HS-Tree) from which all minimal diagnosis sets can be read off [12], [13]. The conflict diagnosis algorithm as introduced by Reiter [13] is based on the repeated application of a conflict detection algorithm [12]. Starting with an initial minimal conflict set, a breadth-first search is conducted by relaxing one constraint of the conflict set at a time. Then, the resulting set of constraints is again checked for consistency and either a solution or a new minimal conflict set is found. Minimal diagnosis sets can be created by following the path from a solution to the root conflict set.

Fig.2 shows the HS-Trees for the three missing invalid tuples of the example. The root nodes are annotated with $\mathcal{O}_{i,j}$ to indicate the corresponding missing invalid tuple. Five different minimal diagnosis sets can be computed: $\triangle_{2,1} = \{c_1\}$, $\triangle_{2,1} = \{c_4\}$, $\triangle_{2,1} = \{c_5\}$, $\triangle_{4,1} = \{c_2\}$ and $\triangle_{5,1} = \{c_2\}$.

It is important to note that the application of one diagnosis set only partially repairs the over-constrained model. Applying $\triangle_{2,1} = \{c_1\}$ only removes a conflict for $\tau_1 \in \mathcal{M}_2$ but conflicts

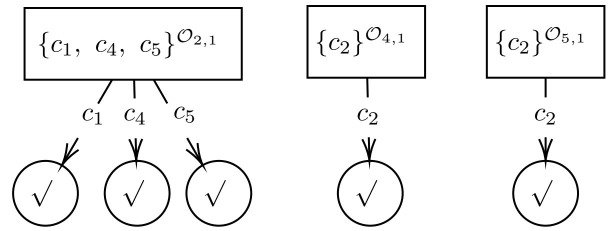

Fig. 2. HS-Trees for missing invalid tuples of the example

for error-constraints $\overline{c_4}$ and $\overline{c_5}$ still exist and require additional relaxation. For each missing invalid tuple $\tau_j \in \mathcal{M}_i$, either one of the computed minimal diagnosis set $\triangle_{i,j}$ must be applied such that τ_j is not missing anymore or error-constraint c_i itself must be relaxed ($\triangle_{i,j} = \{c_i\}$) such that τ_j is discarded. For instance, relaxing c_1 resolves the conflict of $\mathcal{M}_2$ when generating invalid test inputs for error-constraint c_2. But, there are still missing invalid tuples of $\mathcal{M}_4$ and $\mathcal{M}_5$. In contrast, relaxing c_2 resolves two conflicts for error-constraints c_4 and c_5 and discards $\tau_{2,1} \in \mathcal{M}_2$.

Therefore, a set of constraints is required such that their relaxation removes *all* conflicts for all missing invalid tuples.

Definition 8: A **diagnosis hitting set** $\triangle_{HS} \subseteq C^{err} \cup C^{ex}$ is a set of constraints such that the relaxation of all constraints $c \in \triangle_{HS}$ repairs the complete model $\mathcal{T}$. Formally, let $\mathcal{T}$ denote an over-constrained test model. $\triangle_{HS}$ is a diagnosis hitting set if and only if $\mathcal{T}' = \langle P, V, C^{ex} - \triangle_{HS}, C^{err} - \triangle_{HS} \rangle$ is not over-constrained anymore, i.e. no invalid tuples are missing $\forall c_i \in C^{err}$ of $\mathcal{T}' : \mathcal{M}_i = \emptyset$.

A diagnosis hitting set $\triangle_{HS}$ can be composed of diagnosis sets if $\triangle_{HS}$ subsumes at least one diagnosis set for each missing invalid tuple. For the example, three diagnosis sets must be subsumed: $\triangle_{2,1}$, $\triangle_{4,1}$, $\triangle_{5,1}$. For instance, $\triangle_{2,1} = \{c_1\}$, $\triangle_{4,1} = \{c_2\}$, $\triangle_{5,1} = \{c_2\}$ can be composed to $\triangle_{HS} = \{c_1, c_2\}$.

Formally, let $S_{i,j}$ denote the set of all minimal diagnosis sets $\triangle_{i,j}$ for a particular missing invalid tuple $\tau_j \in \mathcal{M}_i$ unified with $\{\triangle_{i,j} = \{c_i\}\}$ representing another minimal diagnosis set that discards $\tau_{i,j}$. For the example, the sets are as follows: $S_{2,1} = \{\{c_1\}, \{c_4\}, \{c_5\}\} \cup \{\{c_2\}\}$, $S_{4,1} = \{\{c_2\}\} \cup \{\{c_4\}\}$ and $S_{5,1} = \{\{c_2\}\} \cup \{\{c_5\}\}$. Further, let $S = S_{i_1,j_1} \times S_{i_2,j_2} \times ...$ denote the entirety of all possible selections of minimal diagnosis sets. Then, each selection $s \in S$ contains exactly one diagnosis set for each missing invalid tuple and each selection can be used to repair the complete model. For the example, the first three columns of Table I depict all possible selections, e.g. $s = \langle \{c_4\}, \{c_4\}, \{c_5\} \rangle$ or $s = \langle \{c_2\}, \{c_2\}, \{c_2\} \rangle$. A selection $s \in S$ can be transformed into a diagnosis hitting set by unifying all contained diagnosis sets. For the example, the fourth column of Table I depicts them.

$$\triangle_{HS}(s) = \bigcup_{\triangle_{i,j} \in s} \triangle_{i,j} \tag{5}$$

The application of each diagnosis hitting set results in a repaired model that is not over-constrained anymore. But to avoid unnecessary changes to the model, only minimal diagnosis hitting sets are further considered.

TABLE I
DIAGNOSIS HITTING SETS FOR THE EXAMPLE (EXCERPT)

$\triangle_{2,1}$	$\triangle_{4,1}$	$\triangle_{5,1}$	$\triangle_{HS}$
$\{c_1\}$	$\{c_2\}$	$\{c_2\}$	$\{c_1, c_2\}$
$\{c_1\}$	$\{c_2\}$	$\{c_5\}$	$\{c_1, c_2, c_5\}$
$\{c_1\}$	$\{c_4\}$	$\{c_2\}$	$\{c_1, c_2, c_4\}$
$\{c_4\}$	$\{c_4\}$	$\{c_5\}$	$\{c_4, c_5\}$
$\{c_2\}$	$\{c_2\}$	$\{c_2\}$	$\{c_2\}$
...	...	...	...

Definition 9: A diagnosis hitting set is **minimal** if and only if no proper subset $\triangle'_{HS} \subset \triangle_{HS}$ is a diagnosis hitting set.

The minimal diagnosis hitting sets for the example are $\triangle_{HS} = \{c_4, c_5\}$ and $\triangle_{HS} = \{c_2\}$.

B. Automatic Relaxation of Conflicting Constraints

Diagnosis hitting sets answer the question *which* constraints to relax but not *how* to relax them. In order to discuss how a constraint must be relaxed, we first explain how constraints are handled in CT and CRT. Afterwards, we discuss appropriate changes to support automatic relaxation.

When generating test inputs , tuples of size t are created and extended until they consist of n parameter-value pairs for all n parameters. Every time a tuple τ is created or extended, constraint handling is involved to check if τ can be extended to contain n parameter-value pairs while satisfying all constraints of a set C, i.e. $\Gamma(\tau, C) = \texttt{true}$. Any extension of τ that does not satisfy all constraints is rejected. In order to check a tuple τ, the test model, all constraints and τ are transformed into a constraint satisfaction problem (CSP).

In general, a CSP consists of three components $\langle X, D, C\rangle$, where 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 [14]. 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. 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 the SAT-solver finds a solution, the tuple τ is accepted and can be further used in test input generation. If no solution exists, the tuple is rejected from further test input generation since one or more constraints are not satisfied.

For the sake of clarity, Fig.3 depicts the internal representation of the example based on integers. In analogy to the transformation of `IPOG-C` [9], the transformation into a CSP is as follows. Each input parameter $p_i \in P$ is represented as a variable $x_i \in X$. The domain of x_i represents the m_i input parameter values V_i as integers $D_{x_i} = \{1, ..., m_i\}$. The

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

Fig. 3. Internal Representation of Exemplary Test Model

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`. All specified constraints are translated to constraints of the CSP accordingly. For instance, $Title \neq 123$ becomes $T \neq 3$. The values of the tuple that is currently being checked are also added as constraints. We refer to them as **tuple-constraints** and a tuple `[Title:Mr, GivenName:John]` translates to $\{T = 1, G = 1\}_\tau$.

$$\begin{aligned} &X = \{T, G, F\} \\ &D = \{D_T = \{1,2,3\}, D_G = \{1,2,3\}, D_F = \{1,2,3\}\} \\ &C = \{T \neq 3, G \neq 3, F \neq 3, T = 2 \Rightarrow G \neq 1, \\ &\qquad T = 1 \Rightarrow G \neq 2\} \cup \{T = 1, G = 1\}_\tau \end{aligned} \tag{6}$$

When using the previously described constraint handling to generate invalid test inputs, the missing invalid tuple $\tau_1 \in \mathcal{M}_2$ (`[GivenName:123]`) evaluates to `false` because it cannot satisfy the remaining constraints. A corresponding diagnosis hitting set $\triangle_{HS} = \{c_4, c_5\}$ can be computed.

However, it remains unclear *how* one of the constraints can be relaxed. Using the previous transformation, one constraint covers several tuples. Error-constraint c_4 of the example specifies two invalid tuples as denoted by $\mathcal{I}_4$ (`[Title:Mrs, GivenName:John]` and `[Title:Mrs, GivenName:123]`). For a minimal conflict set that contains c_4, it is unclear whether the relaxation should remove the first, the second or both tuples.

To provide a finer-grained tuple-level representation of conflict sets, the transformation into CSPs can be adjusted. Instead of translating constraints of the test model directly into constraints of the CSP, a tuple-based CSP can be used where a separate constraint is created for each tuple that is either marked as irrelevant or invalid.

Analogous to error-constraints and invalid tuples ($\mathcal{I}_i$), let $\mathcal{F}_i$ also denote irrelevant tuples that are specified by exclusion-constraint c_i. Then, a separate constraint $c_{i,j}$ is created for each irrelevant or invalid tuple ($\tau_j \in \mathcal{I}_i$ or $\tau_j \in \mathcal{F}_i$) of each constraint $c_i \in C^{ex} \cup C^{err}$ where j indicates the j-th element of $\mathcal{I}_i$ or $\mathcal{F}_i$. Each parameter-value pair (p, v) of an irrelevant or invalid tuple τ_j is translated into a proposition $p = v$, all propositions are combined via logical conjunction and the conjunction is negated such that irrelevant and invalid tuples are avoided.

$$c_{i,j} = \neg\Big(\bigwedge_{(p,v)\in\tau_j} (p = v) \Big) \tag{7}$$

For instance, the constraint $c_4 : T = 2 \rightarrow G = 2$ of the test model (Fig.3), translates into $c_{4,1} = \neg(T = 2 \wedge G = 1)$ and $c_{4,2} = \neg(T = 2 \wedge G = 3)$.

Conflict detection with the tuple-based CSP provides more details to relax a constraint. To resolve the conflict, one constraint c_i must be relaxed such that τ_j is not covered anymore. For instance, a minimal conflict set for $\tau_1 \in \mathcal{M}_2$ (`[GivenName:123]`) is $\mathcal{O}_{2,1} = \{c_{1,1}, c_{4,2}, c_{5,2}\}$. $c_{4,2}$ explains that c_4 must be relaxed such that $\tau_1 \in \mathcal{I}_4$ (`[Title:Mrs, GivenName:123]`) is not covered anymore.

The constraint can be either manually changed or automatically rewritten by adding a logical disjunction for each parameter-value pair of τ_j to c_i.

$$c'_i(\tau_j) = c_i \vee \Big(\bigwedge_{(p,v)\in\tau_j} (p = v) \Big) \quad (8)$$

For instance, error-constraint c_4 can be rewritten as below.

$$\begin{aligned} c'_4 : Title = (Mrs \Rightarrow GivenName = Jane) \\ \vee (Title = Mrs \wedge GivenName = 123) \end{aligned} \quad (9)$$

C. Selection and Application of Diagnosis Hitting Sets

Combining the previously discussed techniques allows completely automated repairs of over-constrained models: Tuple-based CSPs provide all information to automatically relax constraints and diagnosis hitting sets describe sets of constraints that must be relaxed. For instance, the following minimal diagnosis hitting sets can be computed using both techniques: $\triangle_{HS} = \{c_{4,2}, c_{5,2}\}$ and $\triangle_{HS} = \{c_{2,1}\}$.

Focusing on minimal diagnosis hitting sets reduces the number of options, but one out of several minimal diagnosis hitting sets must still be selected. For the example, one has to choose between two. While choosing the cardinality-minimal diagnosis hitting set, i.e. the diagnosis hitting set that contains the fewest constraints, seems appealing, it is not necessarily the best choice. Unfortunately, not every repaired model is *equivalent* to the correct test model in the sense that the generated test suite covers all expected tuples.

To illustrate this, apply $\triangle_{HS} = \{c_{2,1}\}$ to repair the over-constrained test model of the example. Then, $\mathcal{I}_2 = \{$`[GivenName:123]`$\}$ is changed to $\mathcal{I}'_2 = \emptyset$ and no dedicated invalid test input for `[GivenName:123]` must be generated. Instead, two test inputs must be generated to test the invalid value: `[Title:Mrs, GivenName:123]` and `[Title:Mr, GivenName:123]`. Since `[GivenName: 123]` is not excluded from generation of invalid test inputs, not strong invalid test inputs with `[Title:123, GivenName:123]` are possible. This phenomenon is further investigated in the evaluation.

V. Related Work

CRT is implemented in several CT tools: AETG [4], ACTS [6] and PICT [5] include the concept of invalid values. Invalid value combinations are not directly considered. To model them, a workaround is required [3]. In contrast, the approach we proposed in previous work [3] directly considers invalid values and invalid value combinations. Related work in CRT is also further discussed in [3].

Grindal, Offut and Andler [10] survey combination strategies and also discuss coverage criteria for invalid test inputs. Base-choice is another coverage criteria and combination strategy that supports invalid values if the base test input is valid [10]. To model invalid value combinations, the base test input must be adjusted or several base test inputs are required.

In a case study, Wojciak and Tzoref-Brill [15] report on system level CRT with single error t-wise coverage. In another case study [16], we analyze bug reports of a software for life insurances. As a result, only considering invalid values is insufficient for applications with complex input domains.

To repair over-constrained models in CRT, we defined conflicts and missing invalid tuples and applied conflict detection techniques to support manual relaxation [7]. The objective of Gargantini et al. [17], [18] is also to repair constraints of combinatorial test models. Their approach is based on actual execution and they purposely generate test inputs that violate some exclusion-constraints to find exclusion-constraints which are either too weak or too strong. Then, exclusion-constraints are weakened or strengthened to align the test model and SUT.

VI. Evaluation

A. Experiment Design & Setup

In this paper, we propose a semi-automatic technique to repair over-constrained models. Since no comparable work exists yet, the objective of the experiment is to evaluate the general applicability of this technique. We implemented this technique in our CRT prototype , called coffee4j, and applied it to different over-constrained test models. The source code and experiments are available at our companion website[1]. We evaluated the repair technique in two dimensions. We measured the computational overhead and we compared the computed minimal diagnosis hitting sets and evaluated the resulting test inputs.

The experiments are based on 8 benchmark test models. `Addressing` is the running example used throughout this paper. `Registration` is a real-world test model from one of our industry cooperation partners. The other test models originate from [19] and are often used to compare combination strategies. All test models are listed in Table II. The first two columns describe the original test models. The `P & V` column describes the parameter values in exponential notation where v^p refers to p parameters with v values. `Inv. Tuples` describes the invalid tuples as specified by the error-constraints. Here, x^y refers to y invalid tuples for x parameters.

As the original test models are not over-constrained, we added additional error-constraints to artificially create conflicts. Column three describes the additional invalid tuples and the fourth column describes the number of missing invalid tuples. The remaining two columns contain results and are discussed in the next subsection.

B. Results & Discussion

The last two columns of Table II depict the execution times for the test models in milliseconds where the repair technique is applied to both original and modified versions.

The execution times show that the repair technique is a feasible extension with no noteworthy computation overhead, i.e. on average the overhead is 34.75 ms. for the original test models and 831.21 ms for the modified test models. `Banking-1` caused the longest execution times because it specifies 112 invalid tuples that cover all five parameters whereas all other test models specify fewer invalid tuples that cover fewer parameters. The noticeable differences in execution times between original and modified test models is caused

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

TABLE II
TEST MODELS USED FOR EXPERIMENTS AND OVERVIEW OF RESULTS

Name	Original Test Model P & V	Original Test Model Inv. Tuples	Modifications Added Inv. Tuples	Modifications Missing Inv. Tuples	Min. Diagnosis Hitting Sets	Execution Times Original	Execution Times Modified
Addressing	$3^2 2^1$	$2^2 1^3$	2^2	3	2	0.00 ms	1.02 ms
Registration	$6^1 4^4 3^1 2^9$	$2^{17} 1^9$	2^{12}	11	32	6.9 ms	62.14 ms
Banking-1	$4^1 3^4$	5^{112}	$3^2 2^1$	28	16	236.66 ms	6166.74 ms
Banking-2	$4^1 2^{14}$	2^3	$3^2 1^1$	4	4	0.00 ms	2.04 ms
HealthCare-1	$6^1 5^1 3^2 2^6$	$3^{18} 2^3$	1^2	7	4	4.68 ms	29.04 ms
HealthCare-2	$4^1 3^6 2^5$	$5^{18} 3^6 2^1$	$4^6 3^1$	4	8	11.44 ms	54.24 ms
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{31}	$3^2 1^1$	12	10	12.2 ms	127.94 ms
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^{22}	$2^1 1^1$	13	16	6.1 ms	206.54 ms

by the additional computations of conflict and diagnosis sets which is exclusive to over-constrained test models.

To evaluate the computed 92 minimal diagnosis hitting sets, they are applied to the modified test models. Then, the resulting repaired test models are used to generate test inputs which are compared to test inputs from the original test model. The findings are illustrated in Table III which shows a subset of minimal diagnosis hitting sets for `HealthCare-3`. All findings are available at our companion website[1]. Each row of the table relates to another minimal diagnosis hitting set and to the generated test inputs of the repaired test model.

The metric NIT (Not Present Invalid Tuple) counts the number of invalid tuples that are not generated by the repaired test model but are specified by the original test model. For instance, applying $\triangle_{HS} = \{c_{4,2}, c_{5,2}\}$ to the example test model (Fig.1) removes the invalid value combinations `[Title:Mr, GivenName:123]` and `[Title:Mrs, GivenName:123]` and the invalid value `[GivenName:123]` remains. Then, the resulting test suite likely contains only one strong invalid test input with `[GivenName:123]`. When the original test model specifies both invalid value combinations, then at least one is not present in the test suite.

The metric DIT (Duplicate Invalid Tuple) measures the number of invalid tuples that appear more than once. For instance, applying $\triangle_{HS} = \{c_{2,1}\}$ to the example test model removes `[GivenName:123]`. The resulting test suite likely contains two strong invalid test inputs with `[Title:Mr, GivenName:123]` and `[Title:Mrs, GivenName:123]`. If the original test model specifies only one invalid value, then at least one invalid value combination is redundant.

The metric NVTI (Not Valid Test Input) counts the number of valid test inputs generated from the repaired test model that contain invalid tuples as specified by the original test model.

The metric NSITI (Not Strong Invalid Test Input) counts the invalid test inputs generated from the repaired test model that contain more than one invalid tuple as specified by the original test model.

TABLE III
RESULTS FOR HEALTHCARE-3 TEST MODEL (EXCERPT)

$\triangle_{HS}$	Test Suite Size	NIT	RIT	NVTI	NSITI
1	36	0	1	6	2
2	**37**	**0**	**0**	**0**	**0**
3	36	0	1	6	2
4	28	8	1	2	0
5	28	8	1	2	0
6	30	6	2	4	1
...	...	...	...	...	...

The applications of minimal diagnosis hitting sets show that they always result in repaired test models which are not over-constrained anymore. However, not all repaired test models are equivalent to the original test models since the generated test inputs do not cover invalid tuples and do not exclude invalid tuples as specified by the original test model. For each modified test model, exactly one minimal diagnosis hitting set is found that leads to an equivalent test model. Except for `HealthCare-2` where four out of eight minimal diagnosis hitting sets lead to equivalent test models.

The differences between the minimal diagnosis hitting sets can be explained by a relation between the constraint that belongs to the original test model and the constraint that is used in the repaired test model. For instance, error-constraint c_2 of the example model with $\mathcal{I}_2$ = `[GivenName:123]` requires at least one strong invalid test input. At the same time, it forbids the appearance of the invalid value in all other valid and invalid test inputs. In comparison, c_4 with $\mathcal{I}_2$ = `[GivenName:123]` requires at least two strong invalid test inputs. It only forbids the appearance of two invalid value combinations in all other valid and invalid test inputs. Using c_2 while c_4 is correct can lead to invalid tuples that are not present. Using c_4 while c_2 is correct can lead to duplicate invalid tuples as well as to not valid and not strong invalid test inputs. Having another error-constraint c_6 equal to c_2 leads to a conflict where `[GivenName:123]` is specified twice but cannot appear in any invalid test input. The relaxation of either c_2 or c_6 is equivalent and both diagnosis hitting sets result in a repaired test model equivalent to the original test model.

Since the correct test model is unknown, an *oracle* is required to decide which minimal diagnosis hitting sets result in an equivalent test model. As long as no automated oracle exists, some human effort is still required for the selection.

C. Threads to Validity

The results of the comparison might depend on the implementation of the algorithms. To ensure an unbiased implementation of the test input generation, we followed the suggestions for an efficient implementation by Kleine and Simos [20]. For the conflict detection and diagnosis, we explicitly named the used algorithms (QuickXplain[11] and HS-Tree [13]).

The artificial test models do not represent real-world scenarios. However, we based our evaluation on existing benchmark test models, explicitly stated the characteristics and modifications to measure the implications in a controlled environment.

To allow further investigation and replication of the experiments, the prototype and test models are publicly available[1]. The experiments are carried out with an Intel i5 2.20 Ghz CPU and 12 GB of memory. During the execution, resource consumption of other applications may have distorted the results. Therefore, time measurements are based on 50 repetitions.

VII. Conclusion

CRT extends CT to generate separate test suites with either valid and invalid test inputs. Existing CRT approaches rely on additional semantic information that is added to the test model. While the approaches work in general, it is easy to create over-constrained test models. As a consequence, not all specified invalid values and invalid value combinations appear in the test inputs and faults could remain undetected. In previous work, we establish necessary foundations to identify missing invalid tuples and to manually repair over-constrained test models.

In this paper, we extend the previous work by a semi-automatic technique to repair over-constrained test models. Based on minimal conflict sets, we first discuss *which* constraints should be selected for relaxation to completely repair an over-constrained test model. Therefore, we apply conflict diagnosis techniques to identify diagnosis sets to partially repair a test model for a specific missing invalid tuple. Additionally, we introduce diagnosis hitting sets to completely repair a test model for all missing invalid tuples. Afterwards, we discuss *how* the selected constraints can be relaxed. Therefore, we present a finer-grained tuple-based transformations into CSPs which provides all information to automatically relax and rewrite the selected constraints. While the technique is completely automatable, we further argue why semi-automation is preferable as long as no oracle for the selection of diagnosis hitting sets exists.

The presented repair technique is implemented in a prototype and experiments are conducted using benchmark test models. The results show that the repair technique requires only a small overhead for the computation making it applicable in real world. The results also show that the repair technique computes at least one minimal diagnosis hitting set that results in the desired test model. The presented repair technique further reduces the manual effort but still requires some manual work for selecting a diagnosis hitting set.

Because some manual work is still required, we will focus on further techniques in future work that allow a completely automatic generation of test inputs in the presence of over-constrained test models.

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 *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, 2012, pp. 90–97.

[3] K. Fögen and H. Lichter, "Combinatorial robustness testing with negative test cases," in *2019 IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*, 2019, pp. 34–45.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, jul 1997.

[5] J. Czerwonka, "Pairwise testing in real world: Practical extensions to test case generators," in *24th Pacific Northwest Software Quality Conference*, vol. 200, 2006.

[6] L. Yu, Y. Lei, R. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 370–375.

[7] K. Fögen and H. Lichter, "Repairing over-constrained models for combinatorial robustness testing," in *2019 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2019, Sofia, Bulgaria, July 22-26, 2019*, 2019, pp. 177–184.

[8] M. Grindal, J. Offutt, and J. Mellin, "Managing conflicts when using combination strategies to test software," in *18th Australian Software Engineering Conference (ASWEC 2007), April 10-13, 2007, Melbourne, Australia*, 2007, pp. 255–264.

[9] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 242–251.

[10] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

[11] 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, pp. 167–172.

[12] A. Felfernig, S. Reiterer, F. Reinfrank, G. Ninaus, and M. Jeran, "Conflict detection and diagnosis in configuration," in *Knowledge-Based Configuration*, A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, Eds. Boston: Morgan Kaufmann, 2014, pp. 73 – 87.

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

[14] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[15] P. Wojciak and R. Tzoref-Brill, "System level combinatorial testing in practice - the concurrent maintenance case study," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 103–112.

[16] K. Fögen and H. Lichter, "A case study on robustness fault characteristics for combinatorial testing - results and challenges," in *Proceedings of the 6th International Workshop on Quantitative Approaches to Software Quality co-located with 25th Asia-Pacific Software Engineering Conference (APSEC 2018), Nara, Japan, December 4, 2018.*, 2018, pp. 22–29.

[17] 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*, 2016, pp. 49–63.

[18] A. Gargantini, J. Petke, and M. Radavelli, "Combinatorial interaction testing for automated constraint repair," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 239–248.

[19] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 254–264.

[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, pp. 51–67, 2018.