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-
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 \( \tau \) is a set of parameter-value pairs for \( d \) distinct parameters such as \([\text{Title:Mr, GivenName:John}]\). A tuple with \( n \) parameter-value pairs is a test input which can be used to stimulate the SUT. A tuple \( \tau_0 \) covers another tuple \( \tau_a \) if and only if \( \tau_a \) includes all parameter-value pairs of \( \tau_0 \).

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 \( \Gamma(\tau, C) \rightarrow \text{Bool} \) evaluates if a tuple \( \tau \) 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, \( C^{ex} \in T \) is a set of constraints to distinguish between relevant and irrelevant tuples which we denote exclusion-constraints. A tuple \( \tau \) is relevant if it satisfies every exclusion-constraint: \( \Gamma(\tau, C^{ex}) = \text{true} \). A tuple is irrelevant if at least one exclusion-constraint remains unsatisfied: \( \Gamma(\tau, C^{ex}) = \text{false} \).

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 \( C^{err} \in T \)). 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: \( \Gamma(\tau, C^{ex} \cup C^{err}) = \text{true} \). 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: \( \exists c \in C^{err} : \Gamma(\tau, \{c\}) = \text{false} \) and \( \Gamma(\tau, C^{ex} \cup C^{err} \setminus \{c\}) = \text{true} \).

**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 $\text{gen}(P, V, C, t)$ 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 $T$, valid test inputs are generated such that they satisfy all constraints, i.e. $\text{gen}(P, V, C^{ex} \cup C^{err}, t)$. 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 $\neg c_i$) and a set of test inputs is generated that satisfies all constraints including $\neg c_i$ but excluding $c_i$, i.e. $\forall \tau \in C^{err}, \text{gen}(P, V, C^{ex} \cup C^{err} \setminus \{c_i\} \cup \{\neg c_i\}, t)$.

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 $\pi$, all other error-constraints specify tuples that should be excluded. When generating invalid test inputs for error-constraint $\pi$, 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 $\pi$ as $I$. For instance, the following invalid tuples are specified by the example.

\[
I_1 = \{[\text{Title}:123]\} \\
I_2 = \{[\text{GivenName}:123]\} \\
I_3 = \{[\text{FamilyName}:123]\} \\
I_4 = \{[\text{Title}:\text{Mrs}, \text{GivenName}:\text{John}], [\text{Title}:\text{Mrs}, \text{GivenName}:123]\} \\
I_5 = \{[\text{Title}:\text{Mr}, \text{GivenName}:\text{Jane}], [\text{Title}:\text{Mr}, \text{GivenName}:123]\}
\]

Each invalid tuple $\tau \in I_i$ 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 $\pi$, a conflict is a contradiction between error-constraint $\pi$ and some other constraints $C^{err} \setminus \{c_i\} \cup C^{ex}$. The interaction between $\pi$ and some other constraints explicitly or implicitly prevents an invalid tuple $\tau \in I_i$ as specified by $\pi$ 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 $\tau \in I_i$ specified by error-constraint $\pi$ is a missing invalid tuple if and only if a conflict with other error-constraints $C^{err} \setminus \{c_i\}$ or exclusion-constraints $C^{ex}$ prevents it from appearing in any invalid test input.

The set of all missing invalid tuples for error-constraint $\pi$ is denoted as $M_{\pi}$. It is computed by checking for each invalid tuple, if at least one strong invalid test input can be generated. The complexity of computing $M_{\pi}$ is $O(n^2)$, where $n$ is the number of error-constraints.

\[
M_{\pi} = \{\tau | \tau \in I_i : \Gamma(\tau, C^{err} \setminus \{c_i\} \cup C^{ex}) = \text{false} \}
\]

(3) For the example, the missing invalid tuples are as follows.

\[
M_{\pi} = \{[\text{GivenName}:123]\} \\
M_{\pi} = \{[\text{Title}:\text{Mrs}, \text{GivenName}:123]\} \\
M_{\pi} = \{[\text{Title}:\text{Mr}, \text{GivenName}:123]\}
\]

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

Definition 4: A conflict set $O_{i,j} \subseteq C^{err} \setminus \{c_i\} \cup C^{ex}$ is a set of constraints that explains the absence of a missing invalid tuple $\tau_j \in M_{\pi}$. No invalid test input exists that covers $\tau_j$ 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 $O_{i,j}$ is minimal if and only if there exists no proper subset $O'_{i,j} \subset O_{i,j}$ to explain the conflict.

For the running example, the minimal conflict set for the missing invalid tuple $\tau_j \in M_{\pi}$ ($\{[\text{Title}:\text{Mr}, \text{GivenName}:123]\}$) is $O_{4,1} = \{c_2\}$. In contrast, $O_{4,1} = \{c_1, c_2, c_3, c_5\}$ 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 $O_{2,1} = \{c_1, c_4, c_5\}$ explains the absence of the missing invalid tuple $\tau_1 \in M_2$ ([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 $O_{i,j}$ for a given missing invalid tuple $\tau_j \in M_{\pi}$. To find a meaningful...
conflict set, the constraints are partitioned into two disjoint groups [11]: Constraints that are relaxable (denoted as $C$) and background constraints that cannot be relaxed (denoted as $B$). If no solution exists for the constraints $C \cup B$, the model is over-constrained. A proper subset $R \subset C$ is a relaxation if and only if a solution exists for $R \cup B$. However, no relaxation exists if $B$ is inconsistent. A subset of constraints $O \subseteq C$ denotes a conflict set $O$ if and only if no solution exists for $O \cup B$ while $B$ is consistent. Since the conflict set should explain why an invalid tuple $\tau \in M_i$ is missing, $\tau$ should not be relaxed. In contrast, the remaining exclusion- and error-constraints are potentially too strict and should be relaxed: $C = C^{err} \setminus \{c_i\} \cup C^{ex}$.

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 $B$ is consistent and $C \cup B$ has no solution, a diagnosis set $\Delta \subseteq C$ is a set of constraints such that $B \cup C - \Delta$ is consistent.

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

Technically, all constraints $(\Delta_{i,j} = C^{err} \setminus \{c_i\} \cup C^{ex})$ 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 $\Delta_{i,j}$ is minimal if and only if no proper subset $\Delta'_{i,j} \subset \Delta_{i,j}$ 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: $\Delta_{2,1} = \{c_1\}$, $\Delta_{2,1} = \{c_4\}$, $\Delta_{2,1} = \{c_5\}$, $\Delta_{4,1} = \{c_2\}$ and $\Delta_{5,1} = \{c_2\}$.

It is important to note that the application of one diagnosis set only partially repairs the over-constrained model. Applying $\Delta_{2,1} = \{c_1\}$ only removes a conflict for $\tau_1 \in M_4$ but conflicts for error-constraints $\tau_2$ and $\tau_3$ still exist and require additional relaxation. For each missing invalid tuple $\tau_j \in M_i$, either one of the computed minimal diagnosis set $\Delta_{i,j}$ must be applied such that $\tau_j$ is not missing anymore or error-constraint $c_i$ itself must be relaxed ($\Delta_{i,j} = \{c_i\}$) such that $\tau_j$ is discarded. For instance, relaxing $c_1$ resolves the conflict of $M_2$ when generating invalid test inputs for error-constraint $c_2$. But, there are still missing invalid tuples of $M_4$ and $M_5$. In contrast, relaxing $c_2$ resolves two conflicts for error-constraints $c_4$ and $c_5$ and discards $\tau_{2,1} \in M_2$.

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

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

Formally, let $S_{i,j}$ denote the set of all minimal diagnosis sets $\Delta_{i,j}$ for a particular missing invalid tuple $\tau_j \in M_i$ unified with $\{\Delta_{i,j} = \{c_i\}\}$ representing another minimal diagnosis set that discards $\tau_j$. For the example, the sets are as follows: $S_{2,1} = \{\{c_1\}, \{c_4\}, \{c_5\}\} \cup \{\{c_2\}\}$, $S_{4,1} = \{\{c_2\}\} \cup \{\{c_1\}\}$ and $S_{5,1} = \{\{c_2\}\} \cup \{\{c_5\}\}$. Further, let $S = S_{i,j_1} \times S_{i,j_2} \times \ldots$ 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. $s = \{\{c_4\}, \{c_4\}, \{c_5\}\} = \{\{c_2\}, \{c_2\}, \{c_5\}\}$. 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.

$$\Delta_{HS}(s) = \bigcup_{\Delta_{i,j} \in s} \Delta_{i,j}$$

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

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

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

In general, a CSP consists of three components $(X, D, C)$, 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 $\tau$ 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 $D_{x_i} = \{1, \ldots, m_i\}$. The parameter $Title$ is represented as the variable $T$ and its values are $D_T = \{1,2,3\}$. Variable $G$ represents $\text{GivenName}$ and $F$ represents $\text{FamilyName}$. 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 $[\text{Title:Mr, GivenName:John}]$ translates to $\{T = 1, G = 1\}$.

\[
X = \{T, G, F\} \\
D = \{D_T = \{1,2,3\}, D_G = \{1,2,3\}, D_F = \{1,2,3\}\} \\
C = \{T \neq 3, G \neq 3, F \neq 3, T = 2 \Rightarrow G \neq 1, \}
\]

When using the previously described constraint handling to generate invalid test inputs, the missing invalid tuple $\tau_i \in M_2 ([\text{GivenName:123}])$ evaluates to $\text{false}$ because it cannot satisfy the remaining constraints. A corresponding diagnosis hitting set $\Delta_{HS} = \{c_4,c_5\}$ 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 $\mathcal{I}_4 ([\text{Title:Mrs, GivenName:John}]$ and $\text{[Title:Mrs, GivenName:123]}$). 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 ($\mathcal{I}_c$), let $\mathcal{F}_c$ 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 $\tau_j \in \mathcal{I}_c$ or $\tau_j \in \mathcal{F}_c$ of each constraint $c_i \in C^{\text{ex}} \cup C^{\text{irr}}$ where $j$ indicates the j-th element of $\mathcal{I}_c$ or $\mathcal{F}_c$. Each parameter-value pair $(p, v)$ of an irrelevant or invalid tuple $\tau_j$ 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.

\[
c_{i,j} = \neg \left( \bigwedge_{(p, v) \in \tau_j} (p = v) \right) \tag{7}
\]

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

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 $\tau_j$ is not covered anymore. For instance, a minimal conflict set for $\tau_1 \in \mathcal{I}_2 ([\text{GivenName:123}])$ is $\mathcal{O}_{2,1} = \{c_{1,1}, c_{4,1}, c_{4,2}\}$. $c_{4,2}$ explains that $c_4$ must be relaxed such that $\tau_1 \in \mathcal{I}_4 ([\text{Title:Mrs, GivenName:123}])$ 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 $\tau_j$ to $c_i$.

![Fig. 3. Internal Representation of Exemplary Test Model](image-url)
that must be relaxed. For instance, the following minimal
straints and diagnosis hitting sets describe sets of constraints

c_4 (\tau_j) = c_4 \lor \bigwedge_{(p,v) \in \tau_j} (p = v) \tag{8}

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

c_4 : Title = (\text{Mrs} \Rightarrow \text{GivenName} = \text{Jane})
\lor (\text{Title} = \text{Mr} \land \text{GivenName} = 123) \tag{9}

C. Selection and Application of Diagnosis Hitting Sets

Combining the previously discussed techniques allows com-
pletely automated repairs of over-constrained models: Tuple-
based CSPs provide all information to automatically relax con-
straints and diagnosis hitting sets describe sets of constraints
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 \(\Delta_{HS} = \{c_{2,1}\}\) to repair the over-constrained test model of the example. Then, \(I_2 = \{\text{GivenName:123}\}\) is changed to \(I_2 = \emptyset\) 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 ex-
cluded 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, Offutt and Andler [10] survey combination strat-
gies 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 con-
licts 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 \(P \land V\) column
describes the parameter values in exponential notation where
\(v^p\) refers to \(p\) parameters with \(v\) values. Inv. Tuples de-
scribes 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 con-
licts. 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 execu-
tion times between original and modified test models is caused

\footnote{https://github.com/coffee4j/apsec-2019}
HealthCare-3

Min. Diagnosis

<table>
<thead>
<tr>
<th>Name</th>
<th>Original Test Model</th>
<th>P &amp; V Inv. Tuples</th>
<th>Added Inv. Tuples</th>
<th>Missing Inv. Tuples</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing</td>
<td>3^2 1</td>
<td>2^1 3</td>
<td>2^2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Registration</td>
<td>6^4 3^3 1^2 0^1</td>
<td>2^1 3</td>
<td>2^2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Banking-1</td>
<td>4^2 3 1^2</td>
<td>5^1 1^2</td>
<td>3^2 1</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Banking-2</td>
<td>4^2 1^4</td>
<td>2^2</td>
<td>3^2 1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>HealthCare-1</td>
<td>6^3 5^2 3^2 2^6</td>
<td>3^1 5^3</td>
<td>3^2 1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>HealthCare-2</td>
<td>4^3 6^0 5^5</td>
<td>3^1 8^0 3^2</td>
<td>4^2 1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>HealthCare-3</td>
<td>6^1 5^4 3^0 2^6 1^6</td>
<td>2^1 2^2</td>
<td>2^1</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>HealthCare-4</td>
<td>7^1 6^5 4^1 3^2 1^3</td>
<td>2^1 2^2</td>
<td>2^1</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \triangle_{HS} = \{c_3, c_2\} \) 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 result 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 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.

### 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. 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


