Generation of Invalid Test Inputs from Over-Constrained Test 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 with invalid test inputs is important to evaluate the robustness of a system. Combinatorial robustness testing is an approach to generate valid and invalid test inputs separately. Unfortunately, it is easy to create over-constrained test models. As a result, not all specified invalid values or invalid value combinations appear in the test suite. Previous work proposed to repair the test model manually or semi-automatically based on conflict detection and diagnosis techniques. In this paper, we extend that work and present a fully-automatic approach that allows to generate invalid test inputs from over-constrained test models based on alternative constraint handling strategies.**

***Index Terms*—Robustness Testing, Combinatorial Testing, Inconsistent Constraint Set, Model-based Diagnosis**

I. Introduction

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. Oftentimes, error-handlers are implemented to make a system robust by appropriately reacting to external faults. But, error-handlers can contain up to three times more faults than normal source code [2]. Therefore, testing for robustness with invalid inputs is important.

Invalid values and invalid value combinations can cause input masking [3]–[8]. Once the SUT evaluates invalid input, the SUT initiates error-handling and responds with an error message. The remaining values and value combinations remain untested as they are *masked*.

Combinatorial robustness testing (CRT) is an extension to combinatorial testing (CT) that separates the generation of valid and invalid test inputs to avoid potential input masking caused by invalid values and invalid value combinations [7], [8]. Similar to CT, parameters and values are modeled and exclusion-constraints are used to exclude irrelevant value combinations. In addition, the test model is enriched with semantic information to mark certain values and value combinations as invalid. Invalid values and invalid value combinations are excluded from valid test inputs. But, they are included in invalid test inputs such that each invalid value and invalid value combination appears in at least one test input with all other values and value combinations being valid.

Two different approaches to CRT exist. Tools like `AETG` [4], `PICT` [5] or `ACTS` [9] support the annotation of single values as invalid. However, the annotation of invalid value combinations requires a workaround [10]. The tool `coffee4j`[1] implements another approach that directly considers invalid value combinations. Therefore, a second set of constraints (error-constraints) is introduced to describe invalid values and invalid value combinations [7].

Unfortunately, it is easy to create over-constrained test models when using CRT with invalid value combinations [10]–[12]. As a consequence, not all specified invalid values and invalid value combinations appear in invalid test inputs and faults may remain undetected.

Previous work discusses how to identify and repair over-constrained test models manually supported by conflict detection techniques [11]. In addition, previous work discusses how to repair test models semi-automatically using conflict diagnosis techniques that compute *diagnosis hitting sets* (DHS) [12]. When applying conflict diagnosis techniques, several DHS are computed. One DHS must be selected and applied to the test model. The selection of a DHS is important. All DHS can be applied to create test models that are not over-constrained. However, not all DHS result in a *correct* test model and faults may remain undetected [12]. Then, valid test inputs may contain invalid values or invalid test inputs may contain more than one invalid value. As a consequence, a potentially manual oracle is required to select a DHS.

In practice, manual and semi-automatic repair could be rejected because it can be perceived as too time-consuming, too costly or too complex. A fully automated approach is desirable to generate test inputs even in the presence of over-constrained test models. Then, the repair can be postponed or even omitted.

Therefore, this paper presents an automatic approach to generate invalid test inputs directly from over-constrained test

[1] https://coffee4j.github.io

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

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

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

models without repairing the test model beforehand. Instead, different constraint handling strategies are presented to deal with conflicts during the generation.

The paper is structured as follows. First, an example illustrates over-constrained test models and repair techniques. Section III and Section IV summarize foundations of CRT and related work. Different constraint handling strategies are discussed in Section V. As an evaluation, we compare the different strategies in Section VI and conclude with a summary.

II. Example

Throughout the paper, we reuse an example from previous work [11], [12]. It is a customer registration service which checks the correct addressing of a customer. Therefore, the entered data must match the intended semantics of the input fields. Invalid values (denoted as `123`) should not be computed and a customer's title should match the gender of the given name. The service should identify invalid inputs and it should return an error message.

A test model for the example is depicted in Figure 1. Error-constraints describe invalid values like `[GivenName:123]` and invalid value combinations like `[Title:Mrs, GivenName:John]`. For instance, 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 other parameter values are evaluated.

To prevent input masking, a combination strategy generates valid test inputs which satisfy all exclusion- and error-constraints [7]. Invalid test inputs are generated and each specified 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.

The depicted test model is over-constrained. Thus, not all specified invalid values or invalid value combinations appear in the test inputs. First, no test input for `[GivenName:123]` of error-constraint c_2 can be found that satisfies all other constraints: The combinations `[Title:Mr, GivenName:123]` and `[Title:Mrs, GivenName:123]` cannot satisfy error-constraints c_4 and c_5. `[Title:123, GivenName:123]` does not satisfy c_1. The invalid value combinations `[Title:Mr, GivenName:123]` and `[Title:Mrs, GivenName:123]` as specified by c_4 and c_5 cannot satisfy error-constraint c_2.

To repair the test model, either error-constraint $\{c_2\}$ or $\{c_4, c_5\}$ must be relaxed [12]. To relax the latter, the constraints can be rewritten to not longer specify `[Title:Mrs, GivenName:123]` and `[Title:Mrs, GivenName:123]`:

$$\begin{aligned} c_4' &: Title = Mrs \Rightarrow GivenName = Jane \\ &\quad \vee (Title = Mrs \wedge GivenName = 123) \\ c_5' &: Title = Mr \Rightarrow GivenName = John \\ &\quad \vee (Title = Mr \wedge GivenName = 123) \end{aligned} \tag{1}$$

In the next section, techniques to identify and semi-automatically repair over-constrained test models are recapitulated. More details can be found in previous work [7], [11], [12]. Afterwards, the direct generation of invalid test inputs from over-constrained test models is discussed.

III. Combinatorial Robustness Testing

A. Basic Definitions

CRT is a black-box test design technique that systematically generates test inputs based on a given test model. It is an extension of CT that generates separate test suites with valid and invalid test inputs [7]. The **test model** is a quadruple $TM = \langle P, V, C^{ex}, C^{err} \rangle$. It contains a set of n input parameters $P = \{p_1, ..., p_n\}$ and each input parameter p_i is associated with a non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$. C^{ex} and C^{err} are sets of constraints which are explained in subsequent paragraphs.

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 τ_a includes all parameter-value pairs of τ_b.

Further on, a **combination strategy** describes how values are selected and combined to test inputs such that a coverage criterion is satisfied [13]. A **coverage criterion** is a condition that must be satisfied by a combination strategy. In the context of CT, coverage criteria are defined in relation to a test model.

Real-world systems often have restrictions in their input domains and certain combinations of parameter values should not be combined [14]. These value combinations are **irrelevant** as they are, for instance, not executable or of no interest. Therefore, irrelevant value combinations should be removed from the test suite.

Constraint handling can be used to exclude irrelevant value combinations [14]. Constraints are explicitly modeled as logical expressions [15]. A function $\Gamma(\tau, C) \rightarrow \texttt{Bool}$ 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. Formally, we denote a set of constraints to distinguish between relevant and irrelevant tuples as **exclusion-constraints** (C^{ex}). A tuple τ is **relevant** if it satisfies every exclusion-constraint, i.e. $\Gamma(\tau, C^{ex}) = \texttt{true}$. A tuple is **irrelevant** if at least one exclusion-constraint remains unsatisfied, i.e. $\Gamma(\tau, C^{ex}) = \texttt{false}$.

Relevant tuples are further partitioned into valid and invalid tuples. **Valid** tuples are relevant and do not contain any invalid

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

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 contains exactly one invalid value or exactly one invalid value combination to prevent that one masks the other.

Valid test inputs are generated to satisfy a coverage criterion like t-wise coverage excluding all values and value combinations that are irrelevant or invalid. Invalid test inputs are generated to satisfy criteria like **single error coverage** [13] or **t-wise error coverage** [4], [5], [7].

Single error coverage is satisfied if each invalid value and each invalid value combination appears in at least one strong invalid test input. The t-wise error coverage criterion is satisfied if each invalid value or invalid value combination is combined to strong invalid test inputs with all valid combinations of t parameter values. The criterion targets faults that are caused by the interaction of an invalid value or an invalid value combination with other parameter values.

In other words, strong invalid test inputs are generated to ensure that one invalid value or invalid value combination is not masked by another invalid value or invalid value combination. Furthermore, t-wise error coverage reduces the risk that invalid values or invalid value combinations are masked by valid values or valid value combinations.

B. Generation of Strong Invalid Test Inputs

Two different CRT approaches exist [10].

The first approach [3]–[5], [9] includes the concept of invalid values to generate strong invalid test inputs. Values are divided into two disjoint subsets to represent valid and invalid values, i.e. $V_i = V_i^{valid} \cup V_i^{invalid}$. Then, 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. To model invalid value combinations, a workaround which combines invalid values and exclusion-constraints [10] is required.

The second approach [7], [10] includes the concept of of invalid values and invalid value combinations. A new group of constraints is introduced to describe them: **error-constraints** (denoted as C^{err}). A relevant tuple is **valid** if all exclusion-constraints and all error-constraints are satisfied: $\Gamma(\tau, C^{err} \cup C^{ex}) = \texttt{true}$. A relevant tuple is **invalid** if all exclusion-constraints are satisfied but at least one error-constraint remains unsatisfied. It is **strong invalid** if exactly one error-constraint remains unsatisfied: $\exists! c \in C^{err}, \Gamma(\tau, \{c\}) = \texttt{false}$ and $\Gamma(\tau, C^{ex} \cup C^{err} \backslash \{c\}) = \texttt{true}$.

Let `gen(P, V, t, C)` be a t-wise combination strategy that generates a set of test inputs for given input parameters and testing strength `t` that satisfy a set of constraints `C`. Valid test inputs are generated such that they satisfy all constraints, i.e. `gen(P, V, t,` $C^{err} \cup C^{ex}$`)`. Strong invalid test inputs are generated 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 test inputs are generated such that all constraints including $\overline{c_i}$ but excluding c_i are satisfied, i.e. $\forall c_i \in C^{err}$, `gen(P, V, t,` $(C^{err} \backslash \{c_i\}) \cup \{\overline{c_i}\} \cup C^{ex}$`)`. Thereby, strong invalid test inputs can be generated for each specified invalid value and invalid value combination.

The following discussions are based on the second approach. But, the concepts can be transferred to the first approach as well.

Error-constraints 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. But, error-constraint $\overline{c_i}$ also specifies a set of **invalid tuples** (denoted as $\mathcal{I}_i$), i.e. invalid values and invalid value combinations. We denote this set as $\mathcal{I}_i$. To distinguish invalid tuples, we use an index τ_j. While each invalid tuple $\tau \in \mathcal{I}_i$ must not appear in any valid test input, each invalid tuple must appear in at least one invalid test input to either satisfy single or t-wise error coverage.

By the example, the following invalid tuples are specified.

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

C. Repair of Over-Constrained Test Models

Unfortunately, strong invalid test inputs cannot be generated if constraints are over-constrained [10], [11]. Contradictions between constraints prevent some invalid values or invalid value combinations from appearing. For further explanations, conflicts and missing invalid tuples are defined [11].

These ideas are based on conflict detection and diagnosis techniques which are general techniques to identify minimal subsets of constraints that are responsible for faulty behaviour and minimal subsets of constraints that must be relaxed to repair the faulty behaviour [16].

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 $\tau \in \mathcal{I}_i$ from being covered by at least one strong invalid test input.

Definition 2: A test model is considered **over-constrained** if and only if at least one conflict 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 some other constraints $(C^{err} \backslash \{c_i\}) \cup C^{ex}$ prevents it from appearing in any strong invalid test input.

The set of all missing invalid tuples for error-constraint $\overline{c_i}$ is denoted as $\mathcal{M}_i \subseteq \mathcal{I}_i$. An invalid tuple $\tau_j \in \mathcal{I}_i$ is missing if $\tau_j \in \mathcal{M}_i$. We assume that $\tau_{j_1} \in \mathcal{I}_i$ and $\tau_{j_2} \in \mathcal{M}_i$ refer to the

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

same invalid tuple when $j_1 = j_2$. $\mathcal{M}_i$ is computed by checking for each invalid tuple, if at least one strong invalid test input can be generated. For the example, the sets are as follows.

$$\begin{aligned}\mathcal{M}_2 &= \{\texttt{[GivenName:123]}\}\\ \mathcal{M}_4 &= \{\texttt{[Title:Mrs, GivenName:123]}\}\\ \mathcal{M}_5 &= \{\texttt{[Title:Mr, GivenName:123]}\}\end{aligned} \tag{3}$$

To explain the absence of a missing invalid tuple, the notion of conflict sets is introduced [11].

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 $\tau_j \in \mathcal{I}_i$. No invalid test input exists that covers τ_j while satisfying all constraints of the conflict set $\mathcal{O}_{i,j}$.

To resolve the conflict, some constraints of the conflict sets must be relaxed [11]. However, more than one conflict can exist for an error-constraint. To repair the test model for error-constraint c_i, all conflicts must be resolved. All constraints that must be relaxed in order to remove all conflicts for error-constraint c_i can be determined via diagnosis sets [12], [16]:

Definition 5: 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 for τ_j between error-constraint $\overline{c_i}$ and some other constraints can be removed by relaxing all constraints 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 fewer constraints.

Definition 6: 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.

Definition 7: A diagnosis set $\triangle_{i,j}$ is **cardinality-minimal** if and only if there exists no other minimal diagnosis set $\triangle'_{i,j}$ for the same missing invalid tuple $\tau_j \in \mathcal{M}_i$ that consists of fewer constraints, i.e. $\nexists \triangle'_{i,j}$ such that $|\triangle'_{i,j}| < |\triangle_{i,j}|$.

IV. Related Work

Yilmaz, Dumlu, Cohen and Porter [6] define the **input masking effect** "... [as] an effect that prevents a test [input] from testing all combinations of input values, which the test [input] is normally expected to test".

Different types of input masking exist. For instance, test inputs are not executable when they cover irrelevant value combinations and all other values and value combinations of the test input are masked.

A different example about testing configurations of software can be used for illustration. Consider the value combination `[Browser:Edge, OS:Linux]` that is irrelevant because `Edge` cannot be executed on `Linux`. Further on, consider a failure that is triggered by `[Zoom:Enabled, Browser:Edge]`. The failure may not be triggered by 2-wise testing with test input `[Zoom:Enabled, Browser:Edge, OS:Linux]`. To overcome this issue, the irrelevant value combination can be avoided by modelling it as an exclusion-constraint.

Alternatively, the perspective on failure-triggering combinations can be changed such that `[Zoom:Enabled, Browser:Edge, OS:Windows]` instead of `[Zoom:Enabled, Browser:Edge]` triggers the failure. In that sense, increasing the testing strength to $t = 3$ is an alternative to introducing exclusion-constraints.

The same effect can be observed with invalid values and invalid value combinations where error-handling causes an *early return* and other values and value combinations remain untested. In previous work, we measured the effectiveness of CT in the presence of error-handling and invalid values [8]. As a result, CT can be effective. But, the required testing strength to reliably detect faults increases with the number of error-handlers and parameters involved in error-handling. Furthermore, we conduct experiments to compare the effectiveness and efficiency of t-wise error coverage with t-wise coverage [7]. Overall, CRT generates effective and smaller test suites compared with CT. But, CRT requires the additional modelling of invalid values and invalid value combinations.

To the best of our knowledge, Sherwood [3] first discusses input masking in the context of invalid values and invalid value combinations. Cohen [4] and Czerwonka [5] introduce the concept of invalid values to combination strategies. However, since invalid value combinations are not directly supported by their combination strategies, we introduce error-constraints [7], [10]. Although, all CRT approaches support the generation of invalid test inputs that satisfy the single error and t-wise error coverage criterion.

To repair over-constrained test models in CRT, we [11] apply conflict detection techniques and propose a repair process to support manual relaxation. Further, we [12] propose a technique where diagnosis hitting sets (DHS) are computed and one DHS is applied to automatically repair a test model. A DHS is a set of constraints such that their relaxation repairs the complete test model. But, typically more than one DHS is constructed for an over-constrained test model. Then, one out of several DHS must be manually selected. While the application of each DHS repairs a test model, not all repaired test models are necessarily *correct*.

For the example, $\text{DHS}^1 = \{c_4, c_5\}$ and $\text{DHS}^2 = \{c_2\}$ can be computed. Selecting DHS^1 results in the repaired test model as described in Section II (Equation 1). In contrast, selecting DHS^2 removes error-constraint c_2 entirely. Then, two test inputs are generated to test `[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 also possible.

A DHS is constructed by combining diagnosis sets for all missing invalid tuples. In this paper, we also use diagnosis sets. But, instead of applying them directly to the over-constrained test model, we adjust the constraint handling used internally by the combination strategy to allow a fully automated generation.

Pill and Wotawa also apply conflict diagnosis techniques to CT. Although their focus is substantially different. They combine CT and conflict diagnosis techniques to derive knowledge-bases that can be used for abductive diagnosis [17], [18]. Further on, they use conflict diagnosis techniques to

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

compute which components of a SUT explain the failure of test inputs [19].

Gargantini et al. [20], [21] also repair constraints of combinatorial test models. However, their approach is not based on conflict diagnosis techniques. Instead, test inputs that violate some exclusion-constraints are purposely generated and executed to find exclusion-constraints which are either too weak or too strong.

Grindal, Offut and Andler [13] discuss coverage criteria for invalid test inputs as part of a survey on combination strategies. Base-choice is another coverage criteria and combination strategy that supports invalid values if the base test input is valid [13]. To model invalid value combinations, the base test input must be adjusted or several base test inputs are required.

Hallé et al. [22] propose a generalization of t-wise testing and introduce a combination strategy for *existential-constraints* which are constraints that must be satisfied by at least one test input. In their approach, one test input may satisfy more than one existential-constraint whereas in CRT, negated error-constraints must be satisfied by separate test inputs. Hence, CRT also satisfies an existential coverage criterion.

V. Constraint Handling Strategies for Over-Constrained Test Models

A. Overview

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 used to check if possible extensions of $\tau' \supset \tau$ satisfy all constraints. Any extension that does not satisfy all constraints is rejected. In order to check a tuple τ, the parameters, value, all constraints and τ are transformed into a constraint satisfaction problem (CSP). Different approaches for the transformation into CSPs exist which we denote as **constraint handling strategies**.

A CSP consists of three components X, D and C [23]. 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. 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.

A solver searches for a solution of the CSP. If the solver finds a solution, the tuple τ is further used in test input generation. If no solution exists, the tuple is rejected since one or more constraints are not satisfied.

In the following, different constraint handling strategies are discussed. The hard constraint handling strategy is recapitulated. Then, soft constraint handling strategies are discussed.

B. Hard Constraint Handling Strategy

Hard constraint handling (HCH) is the default strategy that is used in tools like `ACTS` or `coffee4j`. HCH requires a solution to satisfy all constraints in order to be consistent. For the sake of clarity, Figure 2 depicts the internal representation of the example (Section II, Figure 1). The transformation into a CSP is as follows. Each input parameter p_i is represented as a variable $x_i \in X$. The domain of x_i represents the m_i values V_i as integers $D_{x_i} = \{1, ..., m_i\}$. All specified 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 $T \neq 3$. The values of the tuple are also added as constraints. We refer to them as **tuple-constraints** and a tuple `[Title:Mr, GivenName:John]` translates to $\{T = 1, G = 1\}_\tau$.

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

Fig. 2. Internal Representation of Example Test Model

$$
\begin{aligned}
&X = \{T, G, F\} \\
&D = \{D_T = \{1, 2, 3\}, D_G = \{1, 2, 3\}, D_F = \{1, 2, 3\}\} \\
&C = \{T \neq 3, G \neq 3, F \neq 3, \\
&\quad T = 2 \Rightarrow G = 2, T = 1 \Rightarrow G = 1\} \\
&\quad \cup \{T = 1, G = 1\}_\tau
\end{aligned} \tag{4}
$$

When using the HCH strategy with an over-constrained test model, single error coverage is not satisfied since missing invalid tuples do not appear in test inputs. The combinations `[Title:Mr, GivenName:123]`, `[Title:Mrs, GivenName:123]` and `[GivenName:123]` are specified but cannot appear in any strong invalid test input.

To cover all specified invalid values and invalid value combinations, the HCH strategy requires the tester to remove all conflicts. In this case, the test model can be repaired by relaxing c_4 and c_5 as depicted by Equation 1 in Section II. However, the techniques presented in related work [11], [12] are manual or semi-automatic.

In the following, the soft constraint handling strategies that does not require any manual work is discussed.

C. Soft Constraint Handling Strategy

1) General Idea: The idea of soft constraint handling (SCH) is based on *partial* constraint satisfaction as described by Freuder and Wallace [24]. Instead of requiring all constraints to be satisfied by a solution, a partial solution that satisfies as many constraints as possible is accepted as well.

For the first time, we introduced the idea of SCH in previous work [10] where we sketched it in an overview fashion as a semi-automatic strategy with a threshold that must be manually determined. Here, we present a substantially extension of the initial SCH idea that relies on diagnosis sets to fully automate the strategy.

For SCH, hard- and soft-constraints are distinguished. A solution must satisfy all hard-constraints (denoted as $\mathcal{H}$) but not all soft-constraints (denoted as $\mathcal{S}$). Even though, it is desirable to satisfy as many soft-constraints as possible.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

To take into account that the satisfaction of soft-constraints is optional, each soft-constraint $c_i \in \mathcal{S}$ is *reified*. An additional boolean variable $r_i \in X$ is created for each soft-constraint c_i and c_i is modified to capture whether or not it is satisfied. The variable r_i is `true` if the constraint c_i is satisfied and `false` otherwise, i.e. $r_i \Leftrightarrow c_i$.

For the example, the boolean variable $r_1 \in X$ is added to reify constraint $c_1 : T \neq 3$. The equality operator is used to align the value of r_1 with the satisfaction of $c_1 : r_1 \Leftrightarrow T \neq 3$.

The CSP is transformed into an optimization problem and a utility function $\mathcal{U}$ counts all reified soft-constraints r_i that are satisfied by assignment α. Then, the solver searches for an assignment α that satisfies all hard-constraints and calculates the utility value.

$$\mathcal{U}(\alpha) = \sum_{c_i \in \mathcal{S}} 1 \times \alpha.r_i \tag{5}$$

When checking whether or not to reject a tuple, a binary answer `true` or `false` is required. However, an optimization problem always leads to a solution. Even zero satisfied soft-constraints with a utility value of zero may represent a consistent assignment. Therefore, a threshold η with $0 \leq \eta \leq |\mathcal{S}|$ is introduced to decide whether or not α is acceptable. The threshold is a lower boundary for the utility function $\eta \leq \mathcal{U}(\alpha)$. An assignment is only consistent if at most η soft-constraints remain unsatisfied. In other words, a consistent assignment must satisfy at least $|\mathcal{S}| - \eta$ soft-constraints.

In the following, two strategies are discussed that are based on the idea of soft constraint handling.

2) Basic Soft Constraint Handling Strategy: The translation of parameters, values and constraints that are considered as hard-constraints is analogous to the HCH strategy. Each input parameter p_i is represented as a variable $x_i \in X$ and x_i represents its domain as integers $D_{x_i} = \{1, ..., m_i\}$. Hard- and soft-constraints are distinguished as follows. The tuple-constraints $\{...\}_\tau$ are considered as hard-constraints. Since we are in the process of finding invalid test inputs for error-constraint c_i, the negated error-constraint $\overline{c_i}$ is also considered as a hard-constraint. The remaining constraints are modelled as soft-constraints.

$$\begin{aligned} \mathcal{H} &= \{\overline{c_i}\} \cup \{...\}_\tau \\ \mathcal{S} &= C^{ex} \cup C^{err} \backslash \{c_i\} \end{aligned} \tag{6}$$

The threshold is enforced by a constraint $|\mathcal{S}| - \eta \leq \mathcal{U}(\alpha)$.

An example CSP to generate invalid test inputs for error-constraint $\overline{c_2}$ is shown below. The negation $\overline{c_2}$ is transformed into the hard-constraint $GivenName = 123$ ($G = 3$ in the internal representation). Constraints c_1, c_3, c_4 and c_5 are modelled as soft-constraints. The threshold is enforced by $|\{r_1, r_3, r_4, r_5\}| - \eta \leq (r_1 + r_3 + r_4 + r_5)$.

$$\begin{aligned} X = &\{\ T, G, F\} \cup \{r_1, r_3, r_4, r_5\ \} \\ D = &\{\ D_T, D_G, D_F, D_{R_{1,3,4,5}} = \{0, 1\}\ \} \\ C = &\{\ r_1 \Leftrightarrow (T \neq 3), \\ &\quad r_3 \Leftrightarrow (F \neq 3), \\ &\quad r_4 \Leftrightarrow (T = 2 \Rightarrow G = 2), \\ &\quad r_5 \Leftrightarrow (T = 1 \Rightarrow G = 1), \\ &\quad 4 - \eta \leq (r_1 + r_3 + r_4 + r_5) \\ &\}\ \cup \{\ G = 3\ \} \cup \{...\}_\tau \end{aligned} \tag{7}$$

In addition, an actual value for the threshold η must be determined. A threshold of $\eta = 0$ is identical with the HCH strategy since all four soft-constraints must be satisfied. For a threshold of $\eta = 1$, a solution must only satisfy three out of four soft-constraints. Then, the following combinations could be generated for error-constraint c_2 because one of the other error-constraints may remain unsatisfied: `[Title:Mr, GivenName:123, FamilyName:Doe]`, `[Title:Mrs, GivenName:123, FamilyName:Doe]`, and `[Title:123,GivenName:123, FamilyName:Doe]`.

To fully automate B-SCH, the threshold η must be determined automatically. It is important to use an appropriate threshold. Otherwise, some missing invalid tuples could remain absent or too many constraints may remain unsatisfied.

A cardinality-minimal diagnosis set $\triangle_{i,j}$ describes a minimal number of constraints to relax in order to resolve all conflicts that prevent $\tau_j \in \mathcal{M}_i$ from appearing in a strong invalid test input for error-constraint $\overline{c_i}$. Hence, a strong invalid test input for $\tau_j \in \mathcal{M}_i$ can be generated if $|\triangle_{i,j}|$ constraints may remain unsatisfied.

Since strong invalid test inputs are generated for one error-constraint at a time, the constraint handling strategy and threshold can also be adjusted individually for each error-constraint. Instead of using a global η for all error-constraints [10], individual thresholds η_i can be determined for all error-constraint $\overline{c_i}$. To determine one threshold η_i, the size of cardinality-minimal diagnosis sets $|\triangle_{i,j}|$ for each missing invalid tuple $\tau_j \in \mathcal{M}_i$ must be checked and the maximum size can be used as the threshold.

$$\eta_i = \max_{\tau_j \in \mathcal{M}_i} |\triangle_{i,j}| \tag{8}$$

If no conflict exists for error-constraint $\overline{c_i}$, i.e. if $\mathcal{M}_i$ is empty, the threshold $\eta_i = 0$ is used which is equivalent to the HCH strategy.

3) Diagnostic Soft Constraint Handling Strategy: Using the size of cardinality-minimal diagnosis sets allows a complete automation of the B-SCH strategy with individual thresholds for each error-constraint. But, there are two more problems with B-SCH.

First, all constraints are modelled as soft-constraints. Since soft-constraints may remain unsatisfied, it can result in unnecessary relaxations. Soft-constraints also increase the CSP by introducing an additional indirection and additional variables via reification.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

Second, the threshold still increases the solution space because thresholds for different invalid tuples of the same error-constraint are not distinguished. For instance, error-constraint $\overline{c_4}$ specifies two invalid value combinations. While `[Title:Mrs, GivenName:John]` is not missing and does not require any relaxation, `[Title:Mrs, GivenName:123]` requires a relaxation of small c_2. However, the threshold $\eta_4 = 1$ is used for both invalid value combinations.

The diagnostic soft constraint handling (D-SCH) strategy exploits the information from the diagnosis sets and to adjust the constraint handling for each invalid tuple individually. Both improvements are addressed in the following.

Minimize Number of Soft-Constraints: It is desirable to have as few soft-constraints as possible. To distinguish hard- and soft-constraints, we use the information from minimal diagnosis sets.

For each missing invalid tuple $\tau_j \in \mathcal{M}_i$, a set of d minimal diagnosis sets $MDS_{i,j} = \{\triangle_{i,j}^1, ..., \triangle_{i,j}^d\}$ can be computed. A constraint $c \in C^{ex} \cup C^{err} \backslash \{c_i\}$ that is not contained in any minimal diagnosis set does not require relaxation. It should be modelled as a hard-constraint. A constraint $c \in C^{ex} \cup C^{err} \backslash \{c_i\}$ that is contained in at least one minimal diagnosis set may require relaxation and should be modelled as a soft-constraint.

$$\begin{aligned}\mathcal{H} =& \{\overline{c_i}\} \cup \{...\}_\tau \cup \\ & \{c | c \in C^{ex} \cup C^{err} \backslash \{c_i\} \wedge \nexists \triangle_{i,j} \in MDS_{i,j} : c \in \triangle_{i,j}\} \\ \mathcal{S} =& \{c | c \in C^{ex} \cup C^{err} \backslash \{c_i\} \wedge \exists \triangle_{i,j} \in MDS_{i,j} : c \in \triangle_{i,j}\}\end{aligned} \tag{9}$$

An example CSP to generate invalid test inputs for error-constraint $\overline{c_2}$ is shown below. In the example, $\overline{c_2}$ specifies one invalid value that is also missing. Three minimal diagnosis sets exist $MDS_{2,1} = \{\triangle_{2,1}^1 = \{c_1\}, \triangle_{2,1}^2 = \{c_4\}, \triangle_{2,1}^3 = \{c_5\}\}$. Hence, error-constraint c_3 can be modelled as a hard-constraint while c_1, c_4, c_5 must be modelled as soft-constraints.

$$\begin{aligned}X &= \{\ T, G, F\} \cup \{r_1, r_4, r_5\ \} \\ D &= \{\ D_T, D_G, D_F, D_{R_{1,4,5}} = \{0, 1\}\ \} \\ C &= \{\ r_1 \Leftrightarrow (T \neq 3), \\ &\quad\ r_4 \Leftrightarrow (T = 2 \Rightarrow G = 2), \\ &\quad\ r_5 \Leftrightarrow (T = 1 \Rightarrow G = 1), \\ &\quad\ 3 - \eta \leq (r_1 + r_4 + r_5) \\ &\quad\} \cup \{\ G = 3\ \} \cup \{\ F \neq 3\ \} \cup \{...\}_\tau\end{aligned} \tag{10}$$

Individual Thresholds: To reduce the solution space, each invalid tuple should have an individual threshold. Invalid tuples $\tau_j \in \mathcal{I}_i$ that are not missing ($\tau_j \notin \mathcal{M}_i$) do not require any relaxation. The threshold should be $\eta_{i,j} = 0$ which is equal to the HCH strategy. In contrast, each missing invalid tuples $\tau_j \in \mathcal{M}_i$ should have an individual threshold that corresponds to the size of the cardinality-minimal diagnosis sets.

$$\eta_{i,j} = \min_{\triangle_{i,j} \in MDS_{i,j}} | \triangle_{i,j} | \tag{11}$$

Since error-constraint $\overline{c_2}$ only specifies one invalid value, the resulting CSP is similar to the one shown above (Equation 10). But, error-constraint $\overline{c_4}$ specifies two invalid value combinations of which one is missing and one minimal diagnosis set exists, i.e $MDS_{4,2} = \{\triangle_{4,2}^1 = \{c_2\}\}$. As a consequence, error-constraints c_1, c_3, c_5 can be modelled as a hard-constraint while only c_2 must be modelled as a soft-constraint. In addition, `[Title:Mrs, GivenName:John]` requires a threshold of $\eta_{4,1} = 0$ and `[Title:Mrs, GivenName:123]` required a threshold of $\eta_{4,2} = 1$.

In previous CSPs, the threshold value η was static and determined beforehand (cf. Equation 8). Now, the CSP distinguishes between several threshold values depending on the actual invalid tuple. As shown below, this behaviour can be modelled with implications following the pattern `(<invalid tuple>)` $\Rightarrow \eta =$ `<value>`.

$$\begin{aligned}X &= \{\ T, G, F\} \cup \{r_2\ \} \\ D &= \{\ D_T, D_G, D_F, D_{R_2} = \{0, 1\}\ \} \\ C &= \{\ r_2 \Leftrightarrow (G = 3), \\ &\quad\ (T = 2 \wedge G = 1) \Rightarrow \eta = 0, \\ &\quad\ (T = 2 \wedge G = 3) \Rightarrow \eta = 1, \\ &\quad\ 1 - \eta \leq (r_2) \\ &\quad\} \cup \{\ T = 2 \wedge G \neq 2) \\ &\quad\} \cup \{\ T \neq 3, G \neq 3, F \neq 3, T = 1 \Rightarrow G = 1\ \} \cup \{...\}_\tau\end{aligned} \tag{12}$$

Using the information provided by minimal diagnosis sets, the CSPs can be formulated much more specific to the individual conflicts as shown above. In the following, the three discussed constraint handling strategies are compared.

VI. Evaluation

A. Experiment Design & Setup

In this paper, we propose alternative constraint handling strategies to use in CRT with over-constrained test models when manual [11] or semi-automatic [12] repair is rejected. The objective of this experiment is to compare the alternative strategies with the default strategy HCH and the semi-automatic repair technique of [12]. We extended the `coffee4j` tool, implemented the constraint handling strategies, and applied them to different over-constrained test models. Source code and experiments are available at our companion website[2].

We evaluated the strategies in two dimensions. First, we measured the computational overhead for test input generation and compared the default HCH strategy with B-SCH and D-SCH. Second, we analyzed characteristics of generated test suites generated by HCH, B-SCH and D-SCH and also compared them with the semi-automatic repair technique.

Eight benchmark test models are used for the experiments [12]. `Addressing` is the running example used throughout this paper. `Registration` is a real-world test model from industry. The other test models originate from [25], [26] and are often used to compare combination strategies. All test models are listed in Table I. 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. `Invalid Tuples` describes the

[2] https://github.com/coffee4j/iwct-2020

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

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, additional error-constraints are added to artificially create conflicts. Column three describes the additional invalid tuples and the fourth column describes the number of missing invalid tuples.

B. Results & Discussion

1) Computational Overhead: Table III depicts the measured times for test input generation in milliseconds using the different constraint handling strategies. The test models include the modifications to make them over-constrained. Because otherwise, the soft-constraint handling strategies are almost equal to HCH.

Overall, the generation times show that the alternative constraint handling strategies are a feasible extension with acceptable overhead. HCH is the fastest constraint handling strategy in seven out of eight times. The absolute difference between the slowest and fastest strategy is on average 1313.09 milliseconds and the relative difference is on average 188.49%.

`Banking-1` performs significantly worse than all other test models. The difference between the slowest and fastest strategy is 6332.86 milliseconds (1028.23%). It requires long generation times because it specifies 112 invalid tuples that cover all five parameters of the test model [12]. Using B-SCH or D-SCH involves a check of all 112 invalid tuples for missing invalid tuples.

Without `Banking-1`, the absolute difference is 595.98 milliseconds (68.52%). Since the measured times range from milliseconds to a few seconds, a slowdown by 68.52% or even 188.49% on average makes the strategies still feasible for application. This is especially true when compared to the manual work that of repairing test models.

There is only one test model (`HealthCare-2`) for which HCH is slower than B-SCH and D-SCH. But, the differences are only 394.90 milliseconds (11.32 %) which is the lowest difference measured among all test models. Therefore, we assume this is caused by distortions from other applications or operating system services.

When comparing the computation overhead of B-SCH with D-SCH, the difference is the additional computation of minimal diagnosis sets in the case of D-SCH. In total, the differences between them are very small being 183.38 milliseconds (18.98%) on average. `Banking-1` is the test model that performed the worst with an overhead of 587.20 milliseconds (62.81%) for D-SCH. Without `Banking-1`, the differences are even smaller (107.14 milliseconds, 17.88%).

2) Characteristics of Generated Test Inputs: Generating test inputs from over-constrained test models with the default HCH strategy results in missing invalid tuples. When repairing an over-constrained test model, the repaired test model can be used to generate test inputs. But, the repaired test model is not necessarily equivalent to the hypothetical correct test model [12]. Some missing invalid tuples may still not appear, some may appear more than once and some may even appear in one test input simultaneously. Therefore, we use the following metrics to analyze and compare test inputs which result from either the over-constrained or repaired test model. We compare them with test inputs which result from the hypothetical correct test model.

The metric `NPIT` (Not Present Invalid Tuple) counts the number of invalid tuples that do not appear in any test input despite being modelled by the correct test model.

The metric `RIT` (Redundant Invalid Tuple) counts the number of invalid tuples that do appear in at least two test inputs.

The metric `NSITI` (Not Strong Invalid Test Input) counts the number of test inputs that contain more than one invalid tuple.

Table II depicts characteristics of repaired test models, i.e. the test suite size of the correct test model, the number of DHS that lead to a correct test model and the total number of computed DHS. A subset of characteristics derived from test input generation is shown in Table IV. All results are available online[2]. For each metric, three numbers are presented as `a (b - c)` where `a` is the average number calculated from 30 repetitions with randomly shuffled test models, `b` denotes the minimum and `c` denotes the maximum number if they differ from the average number.

For the `Addressing` test model, four test inputs are generated using the HCH strategy and one invalid tuple is not present. B-SCH and D-SCH result in seven test inputs that include all invalid tuples as specified by the correct test model. However, both SCH strategies lead to some redundant invalid tuples and to some not strong invalid test inputs. Although, D-SCH has a lower RIT and NSITI. When comparing the SCH strategies with the semi-automatic repair strategy, two DHS are computed of which one must be selected (See Table II). One of the two DHS leads to a correct test model with which five test inputs are generated. In comparison with B-SCH and D-SCH, both repaired test models require fewer test inputs. Both have no NPIT, fewer RIT and also on average fewer NSITI. The running example is the only test model where the repair technique performed better than B-SCH and D-SCH.

When considering `HealthCare-1`, the repair technique again leads to a smaller test suite size but it also leads to 3.10 NPITs on average.

These characteristics can be observed among all test models. While the semi-automatic repair technique leads to smaller test suite sizes on average, it requires the selection of the correct test model to ensure zero NPIT [12]. Otherwise, some invalid tuples may remain absent. In contrast, B-SCH and D-SCH have larger test suite sizes and RIT. But, the test suites have zero NPIT which tackles the problem of over-constrained test models and missing invalid tuples. In direct comparison, D-SCH leads to lower RIT and fewer NSITI than B-SCH which also increases the chance of detecting faults.

VII. Threats to Validity

B-SCH and D-SCH are compared with HCH and with the semi-automatic repair technique. The measured times depend on the implementation of the techniques. Therefore,

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

TABLE I
TEST MODELS USED FOR EVALUATION

Name	Original Test Model		Modifications	
	P & V	Inv. Tuples	Added Inv. Tuples	Miss. Inv. Tuples
Addressing	$3^2 2^1$	$2^2 1^3$	2^2	3
Registration	$6^1 4^4 3^1 2^9$	$2^{17} 1^9$	2^{12}	11
Banking-1	$4^1 3^4$	5^{112}	$3^2 2^1$	28
Banking-2	$4^1 2^{14}$	2^3	$3^2 1^1$	4
HealthCare-1	$6^1 5^1 3^2 2^6$	$3^{18} 2^3$	1^2	7
HealthCare-2	$4^1 3^6 2^5$	$5^{18} 3^6 2^1$	$4^6 3^1$	4
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{31}	$3^2 1^1$	12
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^{22}	$2^1 1^1$	13

TABLE II
CHARACTERISTICS OF REPAIRED TEST MODELS

Test Suite Size	# Correct DHS	# All DHS
5	1	2
26	1	32
112	1	16
3	1	4
21	1	4
25	4	8
31	1	10
22	1	16

TABLE III
TIMES FOR TEST INPUT GENERATION

Name	Constr. Handl. Strategies		
	HCH	B-SCH	D-SCH
Addressing	6.84	9.46	11.94
Registration	1002.90	1638.72	1372.02
Banking-1	615.90	6361.56	6948.76
Banking-2	26.14	38.80	42.40
HealthCare-1	201.60	300.62	489.44
HealthCare-2	3882.10	3487.20	3583.56
HealthCare-3	3432.92	4540.62	4402.14
HealthCare-4	1851.46	3575.70	3522.18

TABLE IV
CHARACTERISTICS OF GENERATED TEST INPUTS

Name	Constr. Handl. Strategy	Test Suite Size	# NPIT	# RIT	# NSITI
Addressing	HCH	4	1	0	0
	B-SCH	7	0	1.67 (1-3)	0.97 (0-3)
	D-SCH	7	0	1.37 (1-2)	0.37 (0-1)
	Repair	5.50 (5-6)	0	0.50 (0-1)	0.17 (0-1)
HealthCare-1	HCH	16	7	0	0
	B-SCH	23	0	0.50 (0-1)	6 (0-12)
	D-SCH	23	0	0	0
	Repair	18.5 (16-21)	3.10 (0-7)	0	0

we integrated our technique into the `coffee4j` tool which also includes HCH and the semi-automatic repair technique.

The used test models do not represent real-world scenarios. But, they are derived from existing benchmark test models and the characteristics as well as modifications are public. For additional investigation and replication, the implementation and test models are publicly available[2]. The experiments are carried out with an Intel i5 2.20 Ghz CPU and 12 GB of memory. Resource consumption of other applications may have distorted the results. Therefore, the measurements are average numbers based on 50 repetitions.

To prevent results that are caused by symmetries between test model and combination strategy, we repeated each experiment 30 times with a randomly shuffled test model.

VIII. CONCLUSION

Combinatorial robustness testing (CRT) extends combinatorial testing (CT) and separates the generation of valid and invalid test inputs in order to avoid input masking caused by error-handling. CRT is based on additional semantic information that allows to distinguish valid from invalid values and value combinations. It is implemented in several CT tools. Although, `AETG`, `ACTS` and `PICT` only include the concept of invalid values. In contrast, `coffee4j` uses error-constraints to model invalid values and invalid value combinations. Unfortunately, it is easy to create over-constrained test models when modeling invalid value combinations which results in missing invalid tuples. As a result, faults can remain undetected. Therefore, previous work applied conflict detection and diagnosis techniques to identify and explain conflicts as well as to repair over-constrained test models semi-automatically.

In this paper, we presented a technique to deal with over-constrained test models fully automatic based on conflict diagnosis techniques. Instead of applying diagnosis sets directly to the over-constrained test model, we adjusted the constraint handling used internally by the combination strategy.

Based on the idea of partial constraint satisfaction, a SCH strategy is proposed which allows some constraints to remain unsatisfied during test input generation. To fully automate the SCH strategy, minimal diagnosis sets are computed to determine the required threshold. Two further improvements are presented to minimize the number of soft-constraints and to compute thresholds specific to each missing invalid tuple.

The alternative constraint handling strategies are implemented in `coffee4j` to allow a comparison with the HCH strategy and the semi-automatic repair technique. As an evaluation, benchmark test models are used to generate test inputs and the time of generation as well as characteristics of the generated test suites are analyzed. The results show that the proposed constraint handling strategies are feasible alternatives to the semi-automatic repair technique. The times measured for test input generation indicate an acceptable computational overhead. Besides the advantage of fully automation, the SCH strategies also tackle the main problem of missing invalid tuples as they lead to zero NPIT. In direct comparison with B-SCH, D-SCH also leads to fewer RIT and NSITI.

In future work, we plan to further improve the performance of SCH by relying on minimal forbidden tuples (e.g. [27]) instead of constraint solving.

REFERENCES

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

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038

[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] G. Sherwood, "Effective testing of factor combinations," in *Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC*, 1994, pp. 151–166.

[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] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. A. Porter, "Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 43–66, 2014.

[7] 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.

[8] ——, "An experiment to compare combinatorial testing in the presence of invalid values," in *Proceedings of the 7th International Workshop on Quantitative Approaches to Software Quality co-located with 26th Asia-Pacific Software Engineering Conference (APSEC 2019), Putrajaya, Malaysia, December 2, 2019.*, 2019, pp. 27–36.

[9] 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.

[10] K. Fögen and H. Lichter, "Combinatorial testing with constraints for negative test cases," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, 2018, pp. 328–331.

[11] ——, "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.

[12] ——, "Semi-automatic repair of over-constrained models for combinatorial robustness testing," in *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*, 2019, pp. 110–117.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] I. Pill and F. Wotawa, "Model-based diagnosis meets combinatorial testing for generating an abductive diagnosis model," in *28th International Workshop on Principles of Diagnosis (DX'17), Brescia, Italy, September 26-29, 2017*, 2017, pp. 248–263.

[18] ——, "On using an I/O model for creating an abductive diagnosis model via combinatorial exploration, fault injection, and simulation," in *29th International Workshop on Principles of Diagnosis (DX'18), Warsaw, Poland, August 27-30, 2018*, 2018.

[19] ——, "Exploiting observations from combinatorial testing for diagnostic reasoning," in *30th International Workshop on Principles of Diagnosis (DX'19), Klagenfurt, Austria, November 11-13, 2019*, 2019.

[20] 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.

[21] 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.

[22] S. Hallé, E. La Chance, and S. Gaboury, "Graph methods for generating test cases with universal and existential constraints," in *Testing Software and Systems*, K. El-Fakih, G. Barlas, and N. Yevtushenko, Eds. Springer International Publishing, 2015, pp. 55–70.

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

[24] E. C. Freuder and R. J. Wallace, "Partial constraint satisfaction," *Artif. Intell.*, vol. 58, no. 1-3, pp. 21–70, 1992.

[25] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved metaheuristic search for constrained interaction testing," in *2009 1st International Symposium on Search Based Software Engineering*. IEEE, 2009, pp. 13–22.

[26] 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.

[27] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–9.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 171-180
DOI 10.1109/ICSTW50294.2020.00038