https://doi.org/10.1109/ICSTW.2018.00068

Combinatorial Testing with Constraints for Negative Test Cases

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*—Constraint handling is an important extension of combinatorial testing to exclude irrelevant combinations which could otherwise lead to the input masking effect. A special handling of invalid values is also important because of potential input masking. Unfortunately, existing CT approaches only consider invalid values explicitly. Invalid value combinations are equally important but only indirectly supported . Therefore, we present a concept that allows to specify invalid value combinations as logical expressions to generate negative test cases.**

***Keywords*-Software Testing, Test Generation, Combinatorial Testing, Input Parameter Modelling, Constraint Handling**

I. Introduction

Combinatorial testing (CT) evolved as an efficient approach to reveal interaction faults [1]. Most real-world applications have constraints in their input parameter domains restricting the number of relevant parameter value combinations. If a test case contains values or value combinations that violate constraints, it cannot be executed or the execution aborts. Oftentimes, the first evaluated invalid value leads to an error-handling procedure within the system under test (SuT). The test fails and other input values of the same test case are masked and remain untested [2]. To avoid input masking, CT integrates constraint handling to annotate irrelevant value combinations and to exclude them from test case generation.

Positive test cases are used to check that the system functions as intended and negative test cases to check proper error-handling. To avoid input masking, positive test cases contain no invalid values and cover all valid t-way value combinations [2], [3]. Negative test cases should contain exactly one negative value. Tools like ACTS [4] or PICT [3] support the exclusion of irrelevant value combinations and the generation of negative test cases based on invalid values. They do not directly support negative test cases based on invalid value combinations. A workaround to model them indirectly in ACTS or PICT is introduced in the background section. However, that is oftentimes insufficient. By not considering invalid combinations, a test case can include more than one invalid combination and errors may remain undetected. By modelling them only indirectly, the model loses clarity and expressiveness making mistakes more likely. In practice, negative test cases are often not negative because of a single invalid value but rather because of an invalid value combination.

$$
\begin{array}{ll}
Title: & \{Mr, Mrs\}^{valid} \\
GivenName: & \{John, Jane\}^{valid} \cup \{123\}^{invalid} \\
FamilyName: & \{Doe\}^{valid} \cup \{123\}^{invalid} \\
\hline
c_1 : Title = Mrs \Rightarrow GivenName \neq John & \\
c_2 : Title = Mr \Rightarrow GivenName \neq Jane &
\end{array}
$$

Listing 1. Exemplary IPM with Error-Constraints

Take a customer registration web-service as an example. It validates the input to avoid invalid names and a wrong addressing of men and women. To test the service, Listing 1 depicts an input parameter model with valid and invalid values above the line. Below the line, error-constraints specify valid value combinations, i.e. the correct addressing. The validation is expected to fail for invalid values or invalid combinations that do not satisfy the error-constraints. Not considering error-constraints results in test cases like `(Mr, Jane, 123)` where an invalid value potentially masks an invalid combination.

In this paper, we extend existing constrained CT algorithms. Our concept allows to directly specify invalid values and value combinations as logical expressions to generate negative test cases automatically. We implement it into the IPOG-C algorithm [4] and apply it to a real-world enterprise application.

The paper is structured as follows. In section II, constraints in CT and related work are discussed. Section III describes the concept to generate negative test cases based on hard and soft-constraints. The evaluation is presented afterwards and we conclude with a summary of our work.

II. Background and Related Work

Combinatorial testing is a black box test design technique to create test cases by applying a combination algorithm on a given input parameter model (IPM) [5]. An IPM is represented as a finite set of n input parameters $IPM = \{p_1, ..., p_n\}$ of which every parameter p_i is related to a finite set of $m_i > 0$ values $V_i = \{v_1, ..., v_{m_i}\}$. An unconstrained combinatorial algorithm with a given strength $t \geq 1$ generates a set of complete test cases that covers all t-way parameter combinations.

$$Solve_{Unconstrained}(IPM, t) = \{\tau_1, ..., \tau_k\}$$

A test case is a tuple of values $\tau = \{v_1, ..., v_l\}$ where $v_i \in V_i$ and $1 \leq l \leq n$. A test case τ is complete if it contains a value for every parameter of the IPM ($l = n$). In contrast, incomplete test cases do not contain values for all parameters ($1 \leq l < n$).

2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 2018, pp. 328-331.
doi: 10.1109/ICSTW.2018.00068

https://doi.org/10.1109/ICSTW.2018.00068

Table I
LOGICAL CONNECTIVES AND THEIR NEGATIONS

Connective	Formula	Negation
Proposition	A	$\neg A$
Negation	$\neg A$	A
Disjunction	$A \vee B$	$\neg A \wedge \neg B$
Conjunction	$A \wedge B$	$\neg A \vee \neg B$
XOR	$A \oplus B$	$(\neg A \wedge \neg B) \vee (A \wedge B)$
Implication	$A \Rightarrow B$	$A \wedge \neg B$
Equivalence	$A \Leftrightarrow B$	$(\neg A \wedge B) \vee (A \wedge \neg B)$

Most real-world applications have constraints that restrict their input parameter domains and separate relevant from irrelevant test cases. Constraint handling in CT allows to exclude irrelevant test cases from generation [1]. Exclusion-constraints are either represented as value combinations that must not appear in any test case (forbidden tuples) or as logical expressions (propositional logic) describing conditions that must be satisfied by all test cases [4]. In the latter case, a function $\Gamma(\tau, c) \rightarrow Bool$ evaluates whether a constraint c is satisfied by a test case τ. Table I summarizes the connectives of propositional logic and their negations.

A constrained combinatorial algorithm generates a set of relevant and complete test cases w.r.t. the exclusion-constraints $C^{Ex} = \{c_1, ..., c_n\}$ such that $\forall \tau \in \{\tau_1, ..., \tau_n\}, \forall c \in C^{Ex}, \Gamma(\tau, c) = true$.

$$Solve_{Constrained}(IPM, t, C^{Ex}) = \{\tau_1, ..., \tau_n\}$$

Every time the constrained CT algorithm chooses a new value, it translates the (incomplete) test case into a constraint satisfaction problem (CSP) and checks if a solution exists that satisfies all exclusion-constraints.

Real-world constraints are often too restrictively formulated and a solution that satisfies all constraints cannot be found. Then it is useful to consider soft-constraints as introduced in related work like [6] or [7]. Hard-constraints specify value combinations that must be satisfied by all test cases. Soft-constraints specify value combinations of which a test case *should* satisfy as many as possible. In the latter case, the CSP is transformed into an optimisation problem seeking to satisfy as many constraints as possible.

It is important to check the behaviour of a SuT with both positive and negative scenarios. Positive scenarios focus on valid intended operations of the system as required by its specification. Negative scenarios focus on robustness and error-handling triggered by invalid inputs. Hence, not only relevant from irrelevant test cases should be distinguished, but also positive from negative ones. Conceptually, the values of every parameter p_i are separated into two disjoint subsets to represent valid and invalid values $V_i = V_i^{valid} \cup V_i^{invalid}$. A test case τ is positive if all values are valid: $\forall v_i \in \tau, v_i \notin V_i^{invalid}$. A test case is negative if there is at least one value in the subset of invalid values: $\exists v_i \in \tau, v_i \in V_i^{invalid}$.

Existing tools combine all positive values to a positive test suite of strength *t* [2], [3]. Additionally, negative test cases are created such that every invalid value should be included in test cases where all other values are valid. This leads to more test cases but avoids possible input masking.

Tools like ACTS or PICT conceptually distinguish valid values from invalid ones but do not consider invalid value combinations. However, invalid value combinations must be expressed as well to generate negative test cases. In this work, error-constraints (denoted as C^{Err}) are used to specify and to reason about the validity of a value or value combination. The two constraints in Listing 1 are examples of them. Similar to exclusion-constraints, the evaluation is true in the absence and false in the presence of invalid values or value combinations.

Since ACTS and PICT do not consider error-constraints, a workaround must be used. It is possible to introduce a new input parameter with one valid value and a set of invalid values: $Error : \{ok\}^{valid} \cup \{err_1, ..., err_n\}^{invalid}$. One invalid value is introduced for each error-constraint. Then, each error-constraint is transformed into an exclusion-constraint of the pattern $(error = err_i) \Leftrightarrow \overline{c_i}$ with $\overline{c_i}$ denoting the negation of c_i. Applying the workaround to the example depicted in Listing 1, constraint no. 1 can be modelled as follows.

$$Error = err_1 \Leftrightarrow (Title = Mrs \wedge GivenName = John)$$

Since $Error = ok$ is chosen for all positive test cases, the right-handed side of the constraint must evaluate to false as well. Afterwards, every invalid value err_i is selected for negative test case generation and the corresponding $\overline{c_i}$ must be true.

The workaround is suitable for many cases. However, it increases the complexity of the model and makes mistakes more likely. It reduces the clarity and expressiveness since additional values must be used and constraints must be negated and are required to match a certain pattern. In practice, it becomes even more complex because $a \Leftrightarrow b$ is not supported by the frontends of ACTS and PICT. Instead, the equivalent expression $a \Rightarrow b \wedge b \Rightarrow a$ must be used.

In the following section, we present an alternative approach that supports error-constraints to model invalid value combinations. Furthermore, soft-constraints are integrated to generate negative test cases for over-constrained models.

III. CONSTRAINTS FOR NEGATIVE TEST CASES

A. General Approach

Existing tools generate relevant test cases based on an IPM, a given strength t and a set of exclusion-constraints to exclude irrelevant value combinations. To generate negative test cases, invalid values and invalid value combinations must be expressed as well. In contrast to the workaround, error-constraints are explicitly integrated. Invalid values are modelled as unary error-constraints rather than as a set $V^{invalid}$.

$$Solve^{Negative}_{Constrained}(IPM, t, C^{Ex}, C^{Err}) = \{\tau_1, ..., \tau_n\}$$

For instance, the values of the parameter `FamilyName` are specified as $FamilyName = \{Doe\}$ and $c_4 : FamilyName \neq 123$. The notion of positive and negative test cases is adjusted as follows. Again, a test case τ is relevant if it satisfies all exclusion-constraints. A positive test case is relevant and satisfies all error-constraints: $\forall c \in C^{Err}, \Gamma(\tau, c) = true$. A negative test case is relevant with at least one unsatisfied error-constraint: $\exists c \in C^{Err}, \Gamma(\tau, c) = false$.

2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 2018, pp. 328-331.
doi: 10.1109/ICSTW.2018.00068

https://doi.org/10.1109/ICSTW.2018.00068

Similar to PICT, the generation process starts with positive test cases. Since positive test cases must satisfy all constraints, they can be generated by an existing algorithm that considers all constraints as exclusion-constraints. The negative test cases are created by additional generation runs for one error-constraint $c_i \in C^{Err}$ at a time. In each run, one error-constraint is negated and a new set of error-constraints is created where $\overline{c_i}$ replaces c_i, i.e. $C' = (C^{Err} \setminus \{c_i\}) \cup \{\overline{c_i}\}$.

Applying an existing algorithm to the modified set of constraints generates negative test cases because the negated constraint $\overline{c_i}$ can only be satisfied by invalid values and invalid value combinations. Since only one constraint is negated each time, the test cases contain one invalid value or value combination. To generate all negative test cases, the replacement and generation is repeated for all $|C^{Err}|$ error-constraints.

```
input:  IPM, t, C^Ex, C^Err
output: A set of test cases
  let T+ = solve(IPM, t, C^Err ∪ C^Ex)
  let T- = ∅
  foreach c_i in C^Err
    let ¬c_i = negation of c_i
    let C' = (C^Err\{c_i}) ∪ {¬c_i}
    T- = T- ∪ solve(IPM, t, C' ∪ C^Ex)
  return T+ ∪ T-
```

Listing 2. Generation of Negative Test Cases

Listing 2 shows the generation algorithm. T^+ denotes the set of positive test cases, T^- the set of negative test cases and the function `solve` refers to an existing constrained algorithm.

In the following subsections, the transformation into CSPs with hard and soft constraints are discussed in more detail.

B. Constraint Satisfaction with Hard-Constraints

An important part of constrained CT algorithms is to check the relevance of a test case. The IPM, all constraints and the test case are transformed into a constraint satisfaction problem. A complete test case τ is relevant if all constraints are satisfied, i.e. $\forall c \in C, \Gamma(\tau, c) = true$. An incomplete test case τ is relevant if values can be assigned to all missing parameters so that the resulting complete test case satisfies all constraints.

Similar to IPOG-C, the example of Listing 1 is translated into the subsequent CSP for positive test cases. Each input parameter is represented as a variable $x_i \in X$. The domain of x_i represents the m_i input parameter values as integers $D_{x_i} = \{1, .., m_i\}$. All constraints of the IPM are translated to constraints of the CSP. The parameter `FamilyName` is represented as `F` and its values `Doe` as 1 and `123` as 2.

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

The constraint `FamilyName` $\neq$ `123` becomes $F \neq 2$. Since the relevance of a test case is to be checked, the values of that test case are added as constraints as well. The test case `(Mr, John)` is modelled as $T = 1 \wedge G = 1$ and it is relevant because a solution `(Mr, John, Doe)` exists.

For negative test cases, one error-constraint of the CSP is negated but all other constraints remain unchanged. Error-constraint no. 2 `(FamilyName = 123)` is negated and transformed to $F = 1$. `(Mr, John, 123)` is a possible solution.

C. Constraint Satisfaction with Soft-Constraints

Real-world models are often over-constrained. The constraints are loose enough such that a solver can find positive test cases. But when generating negative test-cases, negated error-constraints often conflict with other constraints.

In fact, it is quite easy to model constraints that conflict when generating negative test cases. Consider a slightly rewritten version of the constraints no. 1 and 2 that use $=$ instead of $\neq$. There is no difference when generating positive test cases. However, negative test cases cannot be found for constraint no. 3. The negation $\overline{c_3}$ conflicts with c_1' and c_2' because no value for `Title` can be found that satisfies all constraints.

$$
\begin{aligned}
&c_1' : Title = Mrs \Rightarrow GivenName = Jane\\
&c_2' : Title = Mr \Rightarrow GivenName = John\\
&c_3 : GivenName \neq 123\\
&c_4 : FamilyName \neq 123
\end{aligned}
$$

In order to generate negative test cases for over-constrained models, they can be manually relaxed such that they are not over-constrained anymore. However, that is a complicated and error-prone task and sometimes not even possible. Another solution is to use soft-constraints instead. The IPM, constraints and test cases are transformed into an optimisation problem to satisfy as many soft-constraints as possible. Therefore, a weight $\omega_i \in \mathbb{Z}^+$ is assigned to every soft-constraint and a utility function sums all soft-constraints satisfied by assignment a.

$$\max Utility(a) = \max \sum_{i=1}^{|C^{soft}|} \omega_i \times c_i^{soft}(a)$$

When utilising a CSP to check if a test case is relevant or not, it is sufficient to search for a complete assignment. However, that is not sufficient when utilising an optimisation problem. An assignment that does not satisfy a single constraint is complete, even with the worst utility value.

Therefore, three improvements are introduced: First, exclusion-constraints are still hard-constraints that must be satisfied by every test case. They do not affect the value of the utility function. Second, the negated error-constraint to enforce invalid value combinations must also be satisfied by every test case. All other error-constraints are modelled as soft-constraints with weights and have an effect on the utility function. Third, a threshold $\eta \in \mathbb{Z}^+$ is introduced to provide a means for deciding whether a test case is relevant or not. The threshold is a lower boundary for the utility function and a test case is only relevant if the utility value is equal or greater than the threshold $\eta \leq Utility(a)$.

An example to generate negative test cases for $\overline{c_3}$ is depicted below. The negated constraint $\overline{c_3}$ is transformed into the hard-constraint $G = 3$. Constraints no. $1'$, $2'$ and 4 are modelled as soft-constraints. The constraints are *reified* and boolean

2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 2018, pp. 328-331.
doi: 10.1109/ICSTW.2018.00068

https://doi.org/10.1109/ICSTW.2018.00068

variables R_x capture the truth values. A variable R_x is true if the constraint c_x is satisfied and false otherwise ($R_x \Leftrightarrow c_x$). In addition, a constraint for the lower boundary of the utility function is introduced. The CSP with a threshold of $\eta = 3$ and an equal weight $\omega = 1$ for all soft-constraints is similar to the previous approach with hard-constraints only.

$$
\begin{aligned}
X &= \{\ T, G, F\} \cup \{R_1, R_2, R_4\ \} \\
D &= \{\ D_T, D_G, D_F, D_{R_{1,2,4}} = \{0, 1\}\ \} \\
C &= \{\ R_1 \Leftrightarrow (T = 2 \Rightarrow G = 2), R_2 \Leftrightarrow (T = 1 \Rightarrow G = 1), \\
&\qquad R_4 \Leftrightarrow (F \neq 2), \\
&\qquad G = 3, \\
&\qquad \eta \leq \omega_1 R_1 + \omega_1 R_2 + \omega_1 R_3 + \omega_1 R_4\ \}
\end{aligned}
$$

IV. Example and First Evaluation

To illustrate the usefulness of our approach, we implemented it into IPOG-C (called IPOG-C$^{\text{Neg}}$) and compared it to PICT. Test cases are generated by both with a strength of $t = 2$. A $Count$-metric is used to compare them. Since PICT only supports invalid values directly, invalid value combinations are not annotated and modelled as if they were valid. Please note, the results of IPOG-C$^{\text{Neg}}$ and PICT would be equal with the workaround applied to the PICT model.

$$Count(\tau) = |C^{Err}| - \sum_{i=1}^{|C^{Err}|} \Gamma(\tau, c_i)$$

The metric counts the number of invalid values and value combinations per test case by subtracting the number of satisfied error-constraints from the total number of error-constraints. A count of zero unsatisfied error-constraints denotes positive test cases. A count of one denotes *strong* negative test cases that avoid input masking. Numbers greater than one denote negative test cases that deviate from that suggestion and may lead to input masking.

The customer registration example is used as a first scenario. The three inputs are validated at the beginning. If they do not succeed, `false` is returned immediately. Otherwise, the customer is registered. Table II lists the generated pairwise test cases. Only invalid values are considered by PICT whereas all error-constraints are considered by IPOG-C$^{\text{Neg}}$. The counts are listed in the #-column. Two test cases generated by PICT respectively have two unsatisfied error-constraints. Suppose the registration is implemented as shown in Listing 3. Then, the underlined statement is never reached due to input masking. The second validation also fails for the test cases which were supposed to check invalid family names.

```
boolean register(String title, given, family){
  if(!isAGivenName(given)) { return false; }
  if(genderOf(title) != genderOf(given)) {
     return false; }
  if(!isAFamilyName(family)) { return false; }
  ...
}
```

Listing 3. Implementation of the Example Customer Registration

Validation rules for a customer management system of an insurance company are used as a second scenario. The IPM consists of 10 parameters, no exclusion-constraints and 15 error-constraints of which 6 are unary. PICT generates 96 test cases whereas IPOG-C$^{\text{Neg}}$ generates 117 test cases. However, the test cases of IPOG-C$^{\text{Neg}}$ have at most one negative value combination per test case. Not a single positive test case is generated by PICT. On average, PICT's test cases have 3.4 unsatisfied error-constraints and a maximum of seven unsatisfied error-constraints. More evaluation work is necessary but the figures demonstrate the overall usefulness of our extension.

Table II
Generated Test Cases

PICT

Title	Given	Family	#
Mrs	Jane	Doe	0
Mrs	John	Doe	1
Mr	Jane	Doe	1
Mr	John	Doe	0
Mrs	John	123	2
Mr	Jane	123	2
Mr	123	Doe	1
Mrs	123	Doe	1

Extended IPOG-C

Title	Given	Family	#
Mrs	Jane	Doe	0
Mrs	John	Doe	1
Mr	Jane	Doe	1
Mr	John	Doe	0
Mr	John	123	1
Mrs	Jane	123	1
Mr	123	Doe	1
Mrs	123	Doe	1

V. Conclusion

Testing negative scenarios is as important as testing positive ones. Existing CT tools ignore irrelevant combinations and provide a special treatment to invalid values. However, they do not consider invalid value combinations.

In this paper, we presented an approach that extends existing constrained algorithms to directly include invalid value combinations. They can be specified using logical expressions and negative test cases are generated such that one error-constraint is unsatisfied. We experienced over-constrained models when generating negative test cases and provided an alternative solving strategy with a threshold and soft-constraints. Further, the concept was implemented and applied in two scenarios concerned with input validation. Even though further evaluation work is necessary, the results are promising and highlight the usefulness of the concept.

References

[1] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 901–924, 2015.

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

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

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

[5] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, 2007.

[6] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pairwise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 129–139.

2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 2018, pp. 328-331.
doi: 10.1109/ICSTW.2018.00068