DOI 10.1109/QRS.2019.00018

Combinatorial Robustness Testing with 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*—Error-handling is an important means to improve the robustness of a system and testing error-handling is crucial to ensure its correctness. In this paper, we argue that error-handling leads to input masking which requires special treatment in for combinatorial testing. Therefore, we propose an extension to combinatorial testing including a robustness fault model and robustness combination strategy. We also provide an evaluation which compares its efficiency to normal combinatorial testing.**

***Keywords*-Software Testing, Test Generation, Combinatorial Testing, Robustness Testing**

I. Introduction

Robustness describes "the degree to which a system or component can function correctly" in the presence of external faults, e.g. invalid inputs or stressful environmental conditions [1]. External faults can have severe impact on a system's robustness as they can propagate to system failures, e.g. abnormal behavior or system crashes [2]. Defensive programming is a well-known means to improve the robustness by implementing error-handlers to appropriately react to error scenarios [3]. Unfortunately, error-handlers can contain three times more faults than normal code [4].

Therefore, testing with invalid inputs is important to stimulate error-handling of the system under test (SUT) and to check for appropriate responses. Invalid inputs contain invalid values, e.g. a string value when a numerical value is expected, or invalid combinations of values, e.g. a begin date which is after the end date.

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

Combinatorial testing (CT) is a well-known black-box test design approach to check whether a SUT conforms to its specification [5]. Parameters and values are modeled as an input parameter model (IPM) [6] and test inputs are generated such that a combinatorial coverage criterion is satisfied.

An example IPM of the customer registration is depicted in Listing 1. Registering with valid input `[Title:Mrs, GivenName:Jane,FamilyName:Doe]` should succeed whereas the invalid input `[Title:Mrs, GivenName:Jane, FamilyName:123]`[1] should be detected and handled.

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

Listing 1. Exemplary IPM with Valid and Invalid Values

In CT, input masking is a phenomenon which the tester must take care of [7]. Negative test cases can lead to input masking because the invalid test input contains at least one invalid value or invalid value combination. Once the SUT starts evaluating the invalid input, the SUT is expected to detect the first external fault and to initiate error-handling by switching from the normal to the exceptional control-flow. Then, all remaining values and value combinations of the test case remain untested as they are *masked*. For instance, the invalid family name in the test input `[Title:Mrs, GivenName:Jane, FamilyName:123]` leads to error-handling *before* the values of the other parameters are evaluated.

Input masking can be prevented by separating the generation of valid and invalid test inputs. Valid test inputs are generated such that they do not contain any invalid values. Invalid test inputs are generated such that each test input contains at least one invalid value or invalid value combination.

However, existing tools for combinatorial test input generation only distinguish valid from invalid values. Invalid value combinations like `[Title:Mrs, GivenName:John]` are not directly supported [8]. This is not sufficient for systems with complex input domains because input domains are often restricted by invalid combinations of values [9]. In previous work [8], we proposed an approach that also supports invalid value combinations. In this paper, we develop a robustness fault model, describe the approach in more detail and compare it to other black-box testing approaches.

The paper is structured as follows. Section II and III summarize foundations and related work. In Section IV, the robustness fault model and combination strategy for invalid test input generation are discussed. Afterwards, the evaluation is presented. We conclude with a summary of our work.

[1]The value `123` represents some invalid value.

DOI 10.1109/QRS.2019.00018

II. Background

A. Basic Terminology

1) Fault, Error, Failure: A **system** is a set of interacting elements organized to achieve one or more stated purposes. It interacts with other systems, e.g. hardware, software or humans [2]. A system provides one or more services and a client is another system that receives a service. The boundary between the system and other systems is called **environment**.

A system delivers a correct service if it conforms to its specification [2]. A **failure** occurs if the delivered service deviates from the correct service. Usually, a failure is the consequence of an **error** which is a difference between a computed, observed or measured value and the correct value [1]. The identified or hypothesized cause of an error is called a **fault** which is the result of a **mistake** made by a human [1], [2]. A fault is either internal or external of a system [2]. **Internal faults** are, for instance, are incorrect program instructions, usually called as bugs. Examples of **external faults** from the system's viewpoint are invalid inputs or unavailable services provided by other systems. External faults must be detected and handled by the system. Otherwise, they can lead to errors which then can propagate to system failures.

2) Robustness: According to IEEE [1], "the degree to which a system or component can function correctly" in the presence of external faults, is called **robustness**. To make a system robust, the system must become fault tolerant with regards to external faults by considering, specifying and implementing appropriate responses [3].

External fault tolerance aims at failure avoidance by detecting errors caused by external faults and by removing them from the system's state [2]. First, **error-detection** is responsible to identify errors caused by external faults. Then, **error-handling** is responsible to eliminate the errors from the system's state. An error-handler implements the instructions to remove a detected error from the system's state. They are specific to certain types of errors and to certain source code regions of the system. Error-handlers are typically separated from the normal instructions [10]. Therefore, **error-signaling** signals the occurrence of an error and it is switched from normal to **exceptional control-flow**, which is the control-flow path from error-signaling to error-handling [2]. **Error-handler selection** denotes the activity of selecting an error-handler at runtime, once an error is signaled.

Programming languages often include language features to structure error-protection [11]. Once the occurrence of an error is signaled, the runtime environment automatically pauses the normal control-flow and searches for an appropriate error-handler. If no appropriate error-handler is found, the runtime environment typically terminates the system which leads to a failure and leaves the system in a potentially invalid state [10]. The system is recovered if the respective error-handler is executed and the normal control-flow may resume.

Error-handling strategies refer to general solutions used to respond to errors [12]. We distinguish two general strategies. The **internal error-removal strategy** attempts to remove the error locally within the system. For instance, retrying requests to a third-party service that did not respond to the initial request, fixing invalid inputs like replacing a comma with a point in floating-point numbers, or simply logging and ignoring the error if appropriate. If the error can be handled within the system, the normal control-flow can resume and continue. If the system cannot remove the error internally, it must be propagated to the client of the called service. Examples are violations of precondition like a string input when numerical input is expected. We denote this as the **error-propagation strategy**. The normal control-flow does not resume and the exceptional control-flow terminates, e.g. it provides an error-message to the client. For robust systems, this should be a controlled termination (aka graceful degradation) which leaves the system in a valid state [2]. Uncontrolled termination, e.g. if no appropriate error-handler was found, is considered a failure.

In programming languages, **exceptions** are a concept to represent errors in the system [10], [13]. Different types of exceptions can be used to represent different types of errors. We distinguish two general groups. **Runtime-defined exceptions** like `NullPointerException` are automatically detected and signaled by the underlying hardware, operating system or runtime environment. **System-defined exceptions** are specific to the system and the system is also responsible to detect and signal them. These are custom and domain-specific exceptions like `InvalidAddressException`.

The entirety of error-detection, -signaling, -handling and -selection implemented to remove errors caused by certain external faults is called **error-protection**. Programming languages typically include `try-catch` and `throws` language features for error-protection [11]. However, the features often conflict with the goals of object-orientation and are a source of internal faults [10]. Internal faults are denoted as **robustness-faults** if they are related to defensive programming and error-protection and if they prevent a correct removal of errors caused by external faults. Failures that are caused by robustness-faults are denoted as **robustness-failures**.

3) Testing: **Robustness testing** is a special kind of black-box testing [1], [14]. It checks whether the system functions correctly in the presence of external faults.

Testing in general is an activity to check properties by stimulating the system with input and observing its response [2]. The stimulus and response consist of values which are called **test input** and **test output**, respectively. Tests are structures by means of test cases which consist of a test input and a **test oracle** [15]. The test input is necessary to induce the desired behavior. The test oracle provides expected results and determines whether the test passes or fails.

Positive test cases focus on valid intended operations of the SUT using valid input values that are within the specified boundaries. **Negative test cases** are used to test the implemented error-handlers. Therefore, invalid values and invalid value combinations are used as inputs. We consider single values or value combinations that are *outside* of the specified boundaries as invalid. Examples are input that is malformed, e.g. a string input when numerical input is expected, and input

DOI 10.1109/QRS.2019.00018

that violates business rules, e.g. a begin date which is *after* the end date. A test input that contains at least one invalid value or one invalid value combinations is called **invalid test input**. A test input that contains exactly one invalid value or one invalid value combinations is called **strong invalid test input**.

B. Combinatorial Testing

1) Overview: Combinatorial testing (CT) is a black-box test design technique where the input is modelled as an **input parameter model** (IPM). The IPM is represented as a set of n input parameters $IPM = \{p_1, ..., p_n\}$ and each input parameter p_i is a non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$. Input parameters and values are abstract and can represent actual input values to stimulate the SUT, configurations to build and run the SUT, or both. Test inputs are generated such that one value is selected for each input parameter.

Let (p_i, v_j) be a parameter-value pair that denotes a selection of value $v_j \in V_i$ for parameter p_i. A **tuple** is a set of parameter-value pairs for s distinct parameters: $\tau = \{(p_{i_1}, v_{j_1}), ..., (p_{i_s}, v_{j_d})\}$. We use a more concise notation `[Title:Mr, GivenName:John]` which equals $\{(p_1, v_1), (p_2, v_1)\}$. A tuple that consists of n parameter-pairs is a complete test input which can be used to stimulate the SUT. A tuple τ_a **covers** another tuple τ_b (denoted as $\tau_b \subseteq \tau_a$) if and only if for every parameter-value pair (p_i, v_j) of τ_b the same parameter-value pair exists in τ_a, i.e. $\forall (p_{i_b}, v_{j_b}) \in \tau_b, \exists (p_{i_a}, v_{j_a}) \in \tau_a$ such that $i_a = i_b \wedge j_a = j_b$.

The generation of test inputs is usually automated. A **combination strategy** describes how values are selected such that a coverage criterion is satisfied [5]. A **coverage criterion** is a condition which must be satisfied by a test suite. In CT, the t-wise coverage criterion is a common criterion. It is satisfied if all value combinations of t parameters appear in at least one test input. Each-choice ($t = 1$), pair-wise ($t = 2$) and exhaustive ($t = n$) are special cases of it.

2) Input Masking Effect: Most real-world systems have restrictions in their input domains. As a result, most IPMs contain combinations of parameter values that should not be combined [16]. These value combinations are **irrelevant** as they are, for instance, not executable or just not of any interest for the test. For instance, in configuration testing a value combination such as `[Browser:Edge, OS:Linux]` cannot be executed. Since such a test input cannot be executed, the other values and value combinations remain untested as long as there is no other test input that includes them. The irrelevant value combination *masks* all other values and value combinations of the same test input. This effect is called input masking [7]: "The **input masking effect** is an effect that prevents a test case from testing all combinations of input values, which the test case is normally expected to test".

3) Exclusion-Constraints: Because of the input masking effect, a tester should identify and exclude irrelevant value combinations from the test suite. Constraint handling is one strategy to prevent irrelevant value combinations in test inputs while still preserving the coverage criterion [16]. Constraints are explicitly modeled and the combination strategy prevents input masking by excluding irrelevant value combinations.

In this work, we rely on logical expressions to model constraints and on constraint satisfaction problems (CSP) to solve them. A function $\Gamma(\tau, C) \rightarrow \texttt{Bool}$ is used to evaluate whether a tuple τ satisfies a set of constraints C. We denote a set of constraints to distinguish relevant from irrelevant tuples as **exclusion-constraints** (C^{ex}). A tuple τ is **relevant** if it satisfies every exclusion-constraint: $\Gamma(\tau, C^{ex}) = \texttt{true}$. A tuple is **irrelevant** if at least one exclusion-constraint remains unsatisfied: $\Gamma(\tau, C^{ex}) = \texttt{false}$.

4) Invalid Test Inputs: Non-executable test inputs with irrelevant value combinations are only one cause of input masking. The same phenomenon can be observed with invalid test inputs which is neglected by most research. We refer to this special case as **invalid input masking**.

In a successful test, the first invalid value or invalid value combination that is evaluated, triggers error-handling and the normal control-flow is paused. If the error-propagation strategy is used, the SUT responds with an error-message and the normal control-flow is not resumed. Then, all other values and value combinations remain untested because they are masked.

To prevent invalid input masking, test inputs for positive and negative test cases must be generated and they must satisfy separate coverage criteria. The tester's expectation is that all valid values must appear in some valid test inputs that, for instance, satisfy the t-wise coverage criterion. All invalid values and invalid value combinations must not appear in any positive test input but each should appear in at least one invalid test input. Therefore, invalid values and invalid value combinations must be annotated with semantic information such that the combination strategy excludes them from valid test inputs but includes them in invalid test inputs.

III. Related Work

Robustness is often tested by automated robustness testing tools [14], [17]. The SUT is stimulated using random values or boundary values based on parameter types in order to crash the SUT. This approach does not use information from domain experts or specification. In comparison, the objective of our combinatorial robustness testing is to check whether business rules are implemented correctly. For instance, an invalid test input like `[Title:Mrs, GivenName:John, FamilyName:123]` is expected to yield an error message but the implementation has a robustness-fault and does accept the invalid test input. The incorrect error-detection is usually not detected by automated robustness testing tools. They focus more on crashes due to runtime-defined exceptions like `NullPointerException`, that are signaled but not handled, to identify robustness faults.

In contrast, black-box testing techniques are able to find missing but required functionality. The *robust* variants of equivalence class testing (ECT) and boundary value testing (BVT) incorporate invalid values to create invalid test inputs [18]. For robust weak ECT and robustness BVT, one invalid value is combined with all other parameter values being valid.

DOI 10.1109/QRS.2019.00018

Robust strong ECT and robust worst-case BVT describe the cross-product of all valid and invalid values.

Base-choice is another coverage criteria and combination strategy which also supports invalid values [5]. Values are annotated with additional semantic information and a base test input with *default* values for each parameter is created. Additional test inputs are created by replacing one value of one parameter at a time. When the base test input is valid, then base-choice coverage subsumes single error coverage. It also has limited support for invalid value combinations: Since one parameter value of the base test input is replaced at a time, an invalid value combination can be created as a combination of a replacing value and other values of the base test input. However, the failure detection capability of base-choice coverage depends on the quality of the base test input [19]. In contrast, failure detection capability of t-wise coverage only depends on the chosen testing strength [20]. Therefore, we focus on t-wise coverage.

Grindal, Offut and Andler [5] survey combination strategies and also discuss coverage criteria for invalid test inputs. In a case study, Wojciak and Tzoref-Brill [21] report on system level CT that includes invalid inputs. In their case, single error coverage is not sufficient because error-handling depends on interactions between invalid and valid values.

CT tools like AETG [22], ACTS [23] and PICT [24] include the concept of invalid values. Values are divided into two disjoint subsets: $V_i = V_i^{valid} \cup V_i^{invalid}$ to distinguish valid from invalid values. The invalid values of each parameter are excluded from t-wise test input generation. Then, each invalid value is combined with valid values of the other parameters such that every test input contains exactly one invalid value.

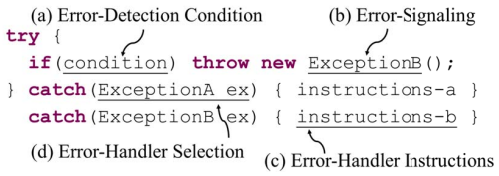
In a case study [9], we analyzed bug reports of a software for life insurances. As a conclusion, only considering invalid values is insufficient for applications with complex input domains. Unfortunately, invalid value combinations are not directly supported by the aforementioned tools. Modeling them as exclusion-constraints is not an option if robustness is a concern. Invalid value combinations would not mask other values and value combinations anymore, but they would remain untested since they are excluded from all test inputs. A workaround to model invalid value combinations exists [8]. It requires to combine invalid values with specific exclusion-constraints. However, the workaround is often insufficient as it drastically reduces clarity and expressiveness.

Our approach differs from the aforementioned tools as it directly supports modeling of invalid value combinations [8]. To the best of our knowledge, it is also the first discussion of a fault model for robustness-faults from a combinatorial perspective and the first comparison between normal t-wise and robustness-based test input generation.

IV. Combinatorial Robustness Testing

A. Robustness Fault Characteristics

1) Overview: Not properly dealing with external faults can affect the system's robustness because they can lead to errors in the system's state which can propagate to robustness-failures



Listing 2. Possible Source Code Regions for Robustness-Faults

[2]. Many modern programming languages have dedicated features to facilitate and ease the implementation of robust systems. However, the features often conflict with the programming language's goals to foster reusability and extensibility [10]. Empirical studies show that they are often not adequately applied and itself a significant cause of robustness-failures [4], [9], [25]–[29]. For instance, the systems analyzed by Sawadpong, Allen and Williams [4] have a three times higher fault density in the implemented error-handlers than in the overall system.

An example of a robustness-fault is *catch and ignore* where the error-handler is empty. Thus, no recovery instructions are executed and the error remains hidden. Another example is *unintended handler action* where the exceptional control-flow is wrong because another, unintended error-handler is executed leading to the execution of wrong error-handling instructions. Please refer to the studies for more details.

The robustness-faults can be grouped into two categories:

2) Configuration-independent robustness-faults: are faults introduced in error-handling which are present in every configuration of the system. To activate them, an invalid input must be used to initiate the fault-containing error-handler.

Based on the aforementioned empirical studies, Listing 2 depicts pseudo-code features for error-handling and highlights source code regions for potential robustness-faults [9].

The source code regions for robustness-faults are as follows.

a) Fault in Error-Detection Condition: A condition used for error-detection is too weak or is completely missing. Then, the error remains undetected.

b) Fault in Error-Signaling: The error is detected but the signal is missing or wrong, e.g. a `NullPointerException` is signaled when a `IllegalArgumentException` would be correct. As a result, not the correct but another error-handler is selected and executed.

c) Fault in Error-Handling: If the error is detected, signaled and the correct error-handler is invoked, the instructions of the error-handler can contain robustness-faults.

d) Fault in Error-Handler Selection: Suppose an error is detected, signaled and the error-handler is correct. Then, the exceptional control-flow can be wrong and not the correct but another error-handler is invoked. For example, let `ExceptionB` be a subtype of `ExceptionA` in Listing 2. Then, the error-handler for `ExceptionB` but the error-handler for `ExceptionA` is unintentionally invoked.

3) Configuration-dependent robustness-faults: Many modern systems like JHipster, Apache or GCC are highly config-

DOI 10.1109/QRS.2019.00018

urable [30], [31]. Features can be enabled or disabled and different implementations of the same feature can be exchanged via configuration parameters at compile-time or at run-time. Typically, these systems can be divided into a common part which is relevant to all configurations and variable features which are relevant only for certain configurations. Different implementation techniques exist [32]. Examples are *conditional compilation* where source code regions are enabled or disabled at compile-time, *dynamic class loading* where classes are chosen and loaded at run-time, *polymorphism* and *design patterns* where implementations are exchanged at runtime, or *aspect-orientation* where meta-programming is used to modify and inject additional instructions at run-time.

Besides robustness-faults present in non-configurable systems, additional **configuration-dependent robustness-faults** may caused by interactions of configuration parameters [30].

Research on exceptional control-flows of highly-configurable systems shows that only few signaled errors are not affected by variable features [33]. Most variable features are either signaling an error, handling an error or they are an intermediate element within the exceptional control-flow. Most of the signaled errors are handled by error-handlers that reside in the same variable feature or in the common part of the system. But, a significant number of errors signaled in variable features remains unhandled [33] which emphasizes the importance of testing configuration-based error-handling.

Listing 3 depicts an example for further illustration. A core banking system has a common part which provides a service to withdraw money from an account. Different implementations of the withdraw functionality exist which are considered variable features. Depending on the account type, then different implementations are chosen at run-time.

Assume `findAccount` signaled a `AccountNotFoundException` for non-existent accounts, the core banking system would crash because no appropriate error-handler exists. This is caused by a configuration-independent robustness-fault because it can be activated with a non-existent account and is present for every configuration.

However, assume a robustness-fault is in the `SavingAccount` feature where the error-signaling condition contains a fault, e.g. plus is used instead of minus. This is caused by a configuration-dependent robustness-fault because it can be activated with the combination of a certain invalid input, i.e. the withdraw amount exceeds the account limit, and a certain configuration that contains the `SavingAccount` class.

```
class CurrentAccount implements Account {
  void withdraw(int amount) {
    if( (balance - amount) < limit )
      throw new AccountNotCoveredException();
    ...
  }
}
class SavingAccount implements Account {
  void withdraw(int amount) {
    if( (balance + amount) < 0 )
      throw new AccountNotCoveredException();
    ...
  }
}
class CoreBankingSystem {
  void withdraw(int accountId, int amount) {
    try {
      Account acc = findAccount(accoundId);
      acc.withdraw(amount);
    } catch(AccountNotCoveredExc. ex) { ... }
  }
}
```

error-detection fault in variable feature

Listing 3. Illustration of Configuration-dependent Robustness-Faults

B. Robustness Fault Model

A **fault model** is a description of hypothesized faults. To observe a failure, the location or locations that contain the fault must be reached (reachability), the state must be incorrect after executing the locations (infection) and the infection must propagate to a service failure (propagation) [15].

In CT, the t-factor fault model assumes that faults are caused by the interaction of parameters [34]. These faults are called **t-factor faults** and t refers to the number of involved parameters. Empirical research [9], [35] indicates that most faults are of factor 1 and 2 whereas no discovered fault involved more than 6 parameters. To satisfy reachability, it is necessary and sufficient to contain the parameter values of the t-factor fault. If the t-factor fault then propagates to a failure, it can be observed by a test oracle [15].

When considering error-protection and the error-propagation strategy, a test input that contains the parameter values of the t-factor fault is not sufficient. The fault can only be reached if there is no invalid input masking caused by other values or value combinations.

Thus, to reach a t-factor fault in the normal control-flow, a test input must contain the corresponding parameter values. Further, the test input must not contain any other invalid values or invalid value combinations that cause invalid input masking.

To reach a configuration-independent robustness-fault, a test input must contain the corresponding invalid value or invalid value combination in order to execute the exceptional control-flow. The values of the other parameters are not relevant as long as they do not contain any irrelevant or invalid values or value combinations that cause an input masking.

In contrast, reaching a configuration-dependent robustness-fault requires the failure-causing configuration to be activated such that the fault becomes reachable. Thus, the strong invalid test input must also contain the specific configuration.

Both, configuration-independent and -dependent robustness-faults can be uniformly described by a **(d,t)-factor robustness-fault**. d denotes the **robustness dimension** which refers to the number of parameters involved in forming an invalid value ($d = 1$) or invalid value combination ($d > 1$). t refers the number of separate configuration parameters involved in forming the fault-activating configuration, i.e. the interaction between invalid value or invalid value combination and other parameters, such that $d + t \leqslant n$.

DOI 10.1109/QRS.2019.00018

C. Robustness Coverage Criteria

A **coverage criterion** is a condition which a test suite must satisfy in order to detect faults of a corresponding fault model.

Combinatorial coverage criteria are defined[2] in terms of the input parameters and values which the tester selects to use [5]. The fault detection effectiveness of a coverage criterion depends on the chosen parameters and values. If certain parameters or values are omitted, related faults cannot be revealed. But, if certain values are combined, the input masking effect can prevent the detection of faults and additional semantic information should be considered. It is important to know which value combinations lead to irrelevant non-executable test inputs and which values and value combinations are expected to initiate error-handlers.

1) Positive t-wise Coverage: To guarantee reachability of a t-factor fault in the normal control flow, we extend the t-wise coverage criterion to exclude all exclusion- and error-tuples.

Therefore, let T^{ex} denote the set of **exclusion-tuples** that represents irrelevant value combinations and let T^{err} denote a set of **error-tuples** that represents invalid values and invalid value combinations. Further on, let $\mathbb{T} = V_1 \times ... \times V_n$ denote the exhaustive set of all possible test inputs. Suppose X_t contains all t-wise parameter value combinations $x \in X_t$ for which a relevant and valid test input $\tau \in \mathbb{T}$ exists, i.e. for which a test input exists that does not cover any exclusion- or error-tuple: $\exists \tau \in \mathbb{T}$ such that $x \subseteq \tau$ and $\forall t \in T^{ex} \cup T^{err} : t \nsubseteq \tau$.

Then, a test suite $\mathbb{S}$ satisfies **positive t-wise coverage** if and only if each tuple $x \in X_t$ is covered by at least one relevant and valid test input $\exists \tau \in \mathbb{S} : x \subseteq \tau$.

2) Negative t-wise Coverage: To guarantee reachability of configuration-independent (d,0)-factor robustness faults, each error-tuple $t_i \in T^{err}$ must appear in some test input. To prevent input masking the test inputs should not cover any exclusion-tuple and should not cover any other error-tuple.

This is manifested as the single error coverage criterion [5], which a test suite $\mathbb{S}$ satisfies if and only if each invalid value and each invalid value combination $t_i \in T^{err}$ appears in at least one strong invalid test input, i.e. $\exists \tau \in \mathbb{S} : t_i \subseteq \tau$ such that $\forall t_j \in T^{ex} \cup T^{err}$ with $j \neq i$ the following $t_j \subseteq \tau$ holds.

However, the criterion is not sufficient to target (d,t)-factor robustness-faults with $t > 0$ since they require an interaction of invalid values or invalid value combinations with t other parameter values.

To guarantee reachability of all (d,t)-factor robustness-faults, we combine the single error and t-wise coverage criterion. Let $\overline{\sigma_i}$ denote the sets of parameter indices that are *not* covered by error-tuple $t_i \in T^{err}$. For instance, for $t_1 =$ `[Family:123]` and $t_2 =$ `[Title:Mrs, GivenName:John]`, the parameter indices are $\overline{\sigma_1} = \{1,2\}$ and $\overline{\sigma_2} = \{3\}$.

Further, suppose $X_t^{\overline{\sigma_i}}$ contains all t-wise parameter value combinations among parameters as indexed by $\overline{\sigma_i}$ such that a relevant and strong invalid test input exist for the combination of each error-tuple t_i and each configuration $x \in X_t^{\overline{\sigma_i}}$.

[2]The definition of coverage criteria follows the formalism introduced by Williams and Probert [36].

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

Listing 4. Exemplary IPM with 5 Error-Constraints

Then, a test suite $\mathbb{S}$ satisfies **negative t-wise coverage** if and only if, for each error-tuple $t_i \in T^{err}$ and for each t-wise parameter value combination $x \in X_t^{\overline{\sigma_i}}$, at least one relevant and strong invalid test input $\tau \in \mathbb{S}$ exists that covers both, i.e. $t_i \subseteq \tau$ and $x \subseteq \tau$.

The negative t-wise coverage criteria subsumes single error coverage. Thus, each test suite $\mathbb{S}$ that satisfies the negative t-wise coverage for any t also satisfies single error coverage.

D. Robustness Input Modelling

To model invalid values and invalid value combinations, we distinguish two different types of constraints. The previously discussed set of exclusion-constraints (denoted as C^{ex}) models value combinations that are irrelevant and should be excluded from test input generation. In addition, **error-constraints** (denoted as C^{err}) further partition relevant test inputs. They allow to separate valid from invalid values and also to separate valid from invalid value combinations.

In contrast to related work, values are not modeled as disjoint sets V_i^{valid} and $V_i^{invalid}$. Only one set of values is used and unary error-constraints describe the validity of values. An example is depicted in Listing 4. The first three error-constraints describe invalid values of three parameters, invalid value combinations are described by c_4 and c_5.

Taking the two types of constraints into account, a tuple τ is **relevant** if it satisfies all exclusion-constraints. A relevant tuple is **valid** if all exclusion-constraints are satisfied and if all error-constraints are satisfied as well: $\Gamma(\tau, C^{err} \cup C^{ex}) = \texttt{true}$. A relevant tuple is **invalid** if all exclusion-constraints are satisfied but at least one error-constraint remains unsatisfied: $\Gamma(\tau, C^{ex}) = \texttt{true}$ and $\Gamma(\tau, C^{err}) = \texttt{false}$. An invalid tuple τ is **strong invalid** if and only if exactly one error-constraint remains unsatisfied: $\exists! c \in C^{err} : \Gamma(\tau, \{c\}) = \texttt{false}$ and $\Gamma(\tau, C^{err} \backslash \{c\}) = \texttt{true}$. Consider the five error-constraints of Listing 4. The tuple `[Title: Mr, GivenName:John, FamilyName:Doe]` is valid as it satisfies all constraints, the tuple `[Title:Mr, GivenName:Jane, FamilyName:Doe]` is invalid because error-constraint c_5 remains unsatisfied.

Error-constraints can be easily transformed into a set of error-tuples (T^{err}) to check for the coverage criteria: A single constraint c_i describes conditions for a non-empty set of k_i parameters. The parameter indices are denoted as $\sigma_i = \{l_1^i, ..., l_{k_i}^i\}$. For instance, error-constraint c_3 of Listing 4 constraints only values of `FamilyName` whereas error-constraint c_4 restricts value combinations of `Title` and `GivenName`. The sets of parameter indices are $\sigma_3 = \{3\}$ and $\sigma_4 = \{1,2\}$, respectively. To obtain the set of error-tuples φ_i described by a

DOI 10.1109/QRS.2019.00018

```
input:  IPM, t+, t-, C^ex, C^err
output: A set of test inputs
  let 𝕊+ = ipog-c(IPM, t+, C^ex ∪ C^err)
  let 𝕊- = ∅
  foreach c_i in C^err
    let c_i' = negation of c_i
    let C' = (C^err\{c_i}) ∪ {c_i'}
    𝕊- = 𝕊- ∪ ipog-neg(IPM, t-, C^ex ∪ C', i)
  return 𝕊+ ∪ 𝕊-
```

Listing 5. ROBUSTA Algorithm

single error-constraint c_i, the cartesian product of all parameter values among the parameter index subset σ_i is computed first. It is then filtered such that each remaining tuple is relevant and invalid, i.e. each remaining tuple does not satisfy the error-constraint c_i but does satisfy all exclusion-constraints. The set of error-tuples (T^{err}) is then simply the union of all φ_i for all error-constraints.

$$\varphi_i \subseteq V_{l_1^i} \times ... \times V_{l_{k_i}^i} \text{ such that}$$
$$\forall \tau \in \varphi_i : \Gamma(\tau, \{c_i\}) = \texttt{false} \wedge \Gamma(\tau, C^{ex}) = \texttt{true}$$

As an example, error-constraint c_5 describes conditions for the parameter-subset $\phi_5 = \{1, 2\}$. The cartesian product $V_1 \times V_2$ is {`[Title:Mr, GivenName:John]`, ..., `[Title:123, GivenName:123]`}. The subset for which c_5 is not satisfiable denotes the set of invalid tuples: φ_5 = {`[Title:Mr, GivenName:John]`}.

E. Robustness Combination Strategy

When considering robustness, both valid and invalid test inputs must be generated. Therefore, we propose a composed combination strategy `ROBUSTA` that delegates the generation of test inputs to other combination strategies. It delegates the generation of valid test inputs to `IPOG-C` and the generation of invalid test inputs to `IPOG-NEG`.

Listing 5 shows the algorithm. t^+ and t^- represent different testing strengths for positive and negative t-wise testing. $\mathbb{S}^+$ denotes the set of valid and $\mathbb{S}^-$ the set of invalid test inputs.

Valid test inputs are generated to satisfy the positive t-wise coverage criterion using t^+ and a combination strategy such as `IPOG-C` [37]. All constraints are treated as exclusion-constraints[3], since valid test inputs must satisfy all of them.

The reused `IPOG-C` combination strategy is explained in [37]. Roughly described, the cartesian product of t parameters is computed and the subset of relevant tuples is used as the initial test suite. Afterwards, the initial test suite is extended by one parameter at a time. In the horizontal extension step, a value $v \in V_i$ of parameter p_i is added to each tuple of the test suite. For uncovered parameter value combinations that involve parameter p_i and (t-1) parameters of $p_1, ..., p_{i-1}$, new test cases are generated in the vertical extension step.

Invalid test inputs which satisfy the negative t-wise coverage are generated afterwards using `IPOG-NEG` with t^-. Invalid test inputs are computed for one error-constraint $c_i \in C^{err}$ at a time. Similar to AETG [22], ACTS [23] and PICT [24], the invalid test inputs are generated such that every error-tuple $t \in T^{err}$ of each $c_i \in C^{err}$ is t^--wise combined with all values of the other parameters ($\overline{\sigma_i}$).

[3]Please note, the distinction between exclusion- and error-constraints only exists for `ROBUSTA`. Within `IPOG-C` and `IPOG-NEG`, all constraints are treated as exclusion-constraints.

```
input:  IPM, t-, C, i
output: A set of test inputs
  let σ_i = {l_1^i, ..., l_{k_i}^i}
  if k_i < t
    extend φ_i with t − k_i additional indices
  let
      𝕊 = subset of V_{l_1^i} × ... × V_{l_{k_i}^i} that satisfies C
  foreach j in {1, ..., n}
    if j not in σ_i
      𝕊 = horizontal-extension(𝕊, p_j)
      if uncovered-tuples-exist(𝕊, p_j)
        𝕊 = vertical-extension(𝕊, p_j)
  return 𝕊
```

Listing 6. IPOG-NEG Algorithm

For example, when generating invalid test inputs for error-constraint c_1 of Listing 4, the error-tuple `[Title:123]` is combined as follows. For a testing strength of $t = 2$, the error-tuple is combined with all valid value combinations of the cartesian product `[GivenName]` × `[FamilyName]`. For $t = 1$, the error-tuple is combined with all valid values of `[GivenName]` and of `[FamilyName]`. A testing strength of $t = 0$ leads to a smaller set of test inputs which contains all error-tuples but does not guarantee any interaction with other parameter values.

Every time invalid test inputs are computed for one error-constraint c_i which is negated and test inputs are generated such that they satisfy the negated error-constraint $\overline{c_i}$ and all other constraints $C^{ex} \cup C^{err} \backslash \{c_i\}$. Since exactly one error-constraint is negated, test inputs only contain one invalid value or one invalid value combination.

The combination strategy for invalid test inputs is a modification of `IPOG-C` which we call `IPOG-NEG` (Listing 6). The modifications ensure that all error-tuples appear and that all error-tuples are t^--wise combined with the other parameter values. `C` is the set of all error- and exclusion-constraints and `i` is the index to identify the negated error-constraint $\overline{c_i}$.

First, the creation of the initial test suite is adjusted. Instead of using t^- arbitrary parameters to compute the cartesian-product, parameters with indices σ_i of error-constraint c_i are used. If σ_i contains less parameter indices than the testing strength $k_i < t$, it is filled up with arbitrary parameter-indices such that $k_i = t$. Since error-constraint c_i is negated, the initial test suite contains all invalid values and invalid value combinations. Afterwards, the initial test suite is extended by all other parameters.

V. Evaluation

A. Overview

In this paper, we propose an approach to CT which considers the presence of error-protection and includes a fault model,

DOI 10.1109/QRS.2019.00018

```
class SystemUnderTest {
  void run(int a1,...,a5, boolean b1,...,b2) {
    if(a1 == 1) { /* exit without failure */
    } else if ...
    } else if(a4 == 1) { fail("f1")
    } else if(a5 == 1) { if(b1) { fail("f2") }
    } else {
      if(a3 == 2 && a4 == 2) { fail("f3") }
    }
  }
}
```

Listing 7. Test Scenario Source Code

$$
\begin{array}{ll}
p_1 : a_1 & V_1 = \{1,2,3,4,5\} \\
p_2 : a_2 & V_2 = \{1,2,3,4,5\} \\
\ldots & \\
p_5 : a_5 & V_5 = \{1,2,3,4,5\} \\
\hline
c_1 : a_1 \neq 1 & \\
c_2 : a_2 \neq 1 & \\
\ldots & \\
c_5 : a_5 \neq 1 &
\end{array}
$$

Listing 8. Test Scenario IPM

coverage criteria and combination strategy. It is considered as an alternative to CT based on t-wise coverage.

The objective of this evaluation is to determine which approach should be used when generating test inputs in the presence of error-protection. Therefore, we compare the proposed ROBUSTA combination strategy to the popular IPOG-C combination strategy.

We developed a prototype[4] which includes ROBUSTA, IPOG-NEG as well as the IPOG-C combination strategy. The implementations are based on the suggestions by Kleine and Simos [38] and are integrated into a Java-based framework for combinatorial testing [39].

The additional indirection of ROBUSTA to distinguish valid and invalid test inputs, i.e. the while-loop with a separate generation for each error-constraint, has an impact on the time required for test input generation. ROBUSTA delegates the generation to IPOG-C once and to IPOG-NEG for each error-constraint. Therefore, the runtime is approximately IPOG-C + $|C^{Err}|$ × IPOG-NEG.

While a faster generation is preferable, longer generation times are usually acceptable since the test inputs only have to be generated once. Therefore, we focus on the number of executed test inputs and the number of detected faults.

B. Experiment Design

A common way for evaluation and comparison of different approaches is to measure the number of detected faults [19]. We measure it by means of fault detection effectiveness [40].

$$\textbf{fault detection effectiveness} = \frac{\text{no. of faults detected}}{\text{no. of all faults in test scenario}}$$

We consider the following SUT characteristics to affect the fault-detection effectiveness of test suites.

a) No. of Parameters: The more parameters a SUT has, the larger the exhaustive input space is. Thus, a smaller portion of the input space is covered by a t-wise test suite. For n parameters with v values, the exhaustive input space is of size v^n while the size of a t-wise test suite is roughly $v^t \log n$ [41].

b) No. of Values: With an increasing number of values, the exhaustive input space of a SUT increases as well and a smaller portion is covered by a t-wise test suite.

[4]Source code and evaluation are available at our companion website https://github.com/coffee4j/qrs-2019.

c) Degree of Invalid Input Masking: The invalid input masking is determined by the number of parameters in error-detection conditions and by the number of values of the affected parameters.

For instance, a SUT with three parameters A, B and C each with two values (1, 2) has an exhaustive input space of $2^3 = 8$. An error-detection condition $A = 1$ would initiate error-handling for four out of eight possible inputs. Thus, a 1-factor fault $C = 2$ covered by four possible inputs would only be reached by two inputs that do not contain $A = 1$. If the three parameters consisted of three values each, the fault would be covered by nine possible inputs and only 3 of them would lead to error-handling. Thus, the reachability increases from $2/4$ to $6/9$. If the error-detection condition of the original example was extended to $A = 1 \wedge B = 1$, error-handling would only be initiated for two inputs and the reachability would increase from $2/4$ to $3/4$. If the error-detection condition $A = 1 \wedge B = 1$ and the fault $A = 2 \wedge B = 2$ share the same parameters, error-handling and fault detection are mutually exclusive. Then, increasing the number of values does not change reachability.

d) Size of Faults: The more parameter values are required to activate a fault, the harder it is to detect it. The 1-factor fault $C = 2$ is covered by four out of eight possible inputs. In contrast, a 2-factor fault $B = 2 \wedge C = 2$ is only covered by two out of eight possible inputs.

C. Experiment Setup

There are three problems with experimentation to evaluate testing approaches [42]: Choosing a representative set of SUTs, choosing a representative set of faults, and choosing a representative set of tests.

To reach the faults, it is important for the IPM to contain all necessary input parameters and values. The reached faults must propagate to failures and the test oracle must reveal the failures. Otherwise, the comparison would be less reliable.

For an experiment, it is important to control the possible and influencing factors. Therefore, we create artificial test scenarios to (1) completely control the size of the input space, (2) completely control the completeness and correctness of the IPM, (3) completely control the number and the characteristics of injected faults, and (4) completely control reachability, infection and propagation of faults to failures.

The test scenarios are described in terms of the aforementioned characteristics. Irrelevant value combinations are not considered in this experiment. Since they were excluded by both approaches, we assume that it would only complicate the experiment but not affect the results.

DOI 10.1109/QRS.2019.00018

For further illustration, Listing 7 depicts the implementation of a test scenario. Suppose we have five input parameters with five values each implemented as integer variables, two binary configuration parameters implemented as boolean variables, and five error-detection conditions consisting of one value each. The separation of normal and exception control-flow is implemented using `if` statements. The exceptional control-flows and different error-detection conditions are implemented by `if` and `else if` whereas the normal control-flow resides in `else`. A fault is activated when the SUT is stimulated with the corresponding parameter values and when the location of the fault is reached which is modeled by `fail()`. The example contains a 2-factor fault in the normal control-flow `f3`, a (1,0)-factor configuration-independent robustness fault `f1` and a (1,1)-factor configuration-dependent robustness fault `f2` which is only reachable when configuration `b1` is enabled.

The corresponding IPM is depicted in Listing 8. The error-constraints are ignored by `IPOG-C` but considered by `ROBUSTA` to separate valid from invalid test inputs.

For a given scenario and IPM, two test suites are generated using `IPOG-C` and `ROBUSTA`. The `run` method is executed for all test inputs and the fault detection effectiveness is computed based on the detected faults.

D. Experiment Scenarios

In this experiment, we use two *base* scenarios which are extended according to the aforementioned characteristics in order to observe their impact on the fault detection effectiveness.

Base scenario 1 consists of five input parameters and five configuration parameters. It includes five error-detection conditions of which each condition is unary and checks a separate input parameter. The faults are triggered by values of the same five input parameters. Thus, there is a mutual exclusion between error-detection and fault detection.

Base scenario 2 consists of ten input parameters and five configuration parameters. It also includes the same five error-detection conditions of which each condition is unary and checks a separate input parameter. However, the included faults are triggered by values of the other five input parameters that are not checked by error-detection conditions. Thereby, we can observe whether the relationship between parameters involved in error-detection and parameters involved in t-factor faults has an impact on the fault detection effectiveness.

Both scenarios have a fixed set of t-factor faults with factors of one to four because these are the most common factors identified in empirical studies [35]. In addition, they also contain a fixed set of (d,t)-factor robustness faults. They have to be defined relative to the error-detection conditions. A (d,0)-factor robustness fault is always reached when the corresponding error-detection condition is satisfied. In contrast, a (d,2)-factor robustness fault is only reached when the error-detection condition is satisfied and the fault-activating configuration is activated. The t-factors range from factor zero to three.

The base scenarios are then extended such that (a) the number of values per parameter increases and (b) the number of parameters increases. Further scenarios are then derived from each extended scenario by increasing the number of parameters involved in the error-detection conditions. The scenarios start with five unary conditions of which each checks a separate parameter. The conditions are extended to check two values, e.g. $a_1 = 1$ is extended to $a_1 = 1 \wedge a_2 = 2$, three and four values, e.g. $a_1 = 1 \wedge a_2 = 2 \wedge a_3 = 3 \wedge a_4 = 4$.

To reduce the effect of accidentally discovered faults, e.g. a 3-factor fault is detected by a 2-wise test suite, and to break the symmetry of the SUT and IPM, i.e. the faults and the parameter values are ordered the same way, the parameters and values of the IPM are randomly shuffled and the experiment is executed ten times.

Table I depicts all scenarios used in the experiment. It uses an exponential notation where x^y refers to y parameters consisting of x values, y t-factor faults of factor x, and y error-detection conditions with x parameters. For $(d,t)^y$-factor robustness-faults, d refers to the number of involved input parameters that form an invalid value ($d = 1$) or an invalid value combination ($d > 1$) which is always similar to the parameters involved in error-detection conditions. t refers to the number of involved configuration parameters necessary to make the fault reachable and y depicts the number of faults.

E. Results & Discussion

To compare `IPOG-C` and `ROBUSTA`, test inputs are generated with different strengths. For `IPOG-C`, the strengths range from one to five. For `ROBUSTA`, the strengths range from `1-0` to `4-3` where the first number represents the positive strength t^+ and the second number represents the negative strength t^-.

All 24 scenarios are evaluated by `IPOG-C` with five different strengths and by `ROBUSTA` with 16 different strengths. Each triple of scenario, combination strategy and strength is computed ten times with randomly shuffled IPM parameters and values. A subset of the resulting 5040 computations is listed in Table II. All results are available on our companion website[4]. The presented sizes of test suites and computed fault detection effectiveness are average numbers of the ten randomized IPMs.

`ROBUSTA` with a strength of `4-3` activates all faults. The result can be expected because the seeded faults are at most 4-factor faults and (4,3)-factor robustness faults. But, it also demonstrates the correctness of the coverage criterion and combination strategy. However, lower testing strength are not sufficient. With a lower positive testing strength like `3-3`, not all 4-factor faults in the normal control-flow are activated. With a lower negative testing strength like `4-2`, not all (4,3)-factor robustness faults are activated.

In comparison, `IPOG-C` does not activate all faults for all scenarios. For the scenarios with unary error-detection conditions (`1-1-1`, `1-2-1`, `1-3-1`, ...), `IPOG-C` with a testing strength of four activates almost all faults. The small deviations in fault detection efficiency, e.g. 0.98 and 0.99 for scenarios `1-1-1`, `1-2-1`, and `2-1-1`, can be explained by invalid input masking. It performs as good as `ROBUSTA` if you compare the sizes of test suites. If you consider the additional effort to model error-constraints, `IPOG-C` is actually preferable because it requires less work than `ROBUSTA`.

DOI 10.1109/QRS.2019.00018

Table I
SCENARIOS USED IN EXPERIMENTS

Scenario	Description	Parameters & Values	t-factor Faults	(d,t)-factor Robustness Faults	Error-Detection
1-1-1	**Base Scenario 1**	$2^5 5^5$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-1-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-1-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-1-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-2-1	**Increase No. of Values**	$3^5 7^5$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-2-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-2-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-2-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-3-1	**Increase No. of Parameters**	$2^{10} 5^{10}$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-3-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-3-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-3-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-4-1	**Increase Values & Parameters**	$3^{10} 7^{10}$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-4-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-4-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-4-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
2-1-1	**Base Scenario 2**	$2^{10} 5^{10}$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
2-1-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
2-1-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
2-1-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
2-2-1	**Increase No. of Values**	$3^{10} 7^{10}$	$1^5 2^5 3^5 4^5$	$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
2-2-2	* Increase Size of Conditions			$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
2-2-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
2-2-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5

However, scenarios with error-conditions consisting of two or more parameters require higher strengths. For a given testing strength, the fault detection effectiveness of `IPOG-C` decreases when the numbers of parameters involved in error-detection increases because a (1,3)-factor robustness fault becomes a (4,3)-factor robustness fault. As an example, the fault detection effectiveness decreases for a strength of four from an average of 0.993 (39.7 of 40 faults are activated) for scenarios with unary conditions to an average of 0.763 (30.5 of 40 faults are activated) for scenarios with 4-wise conditions.

This phenomenon can be explained by considering (d,t)-factor robustness faults as a special kind of t-factor faults. Again, a (d,t_2)-factor[5] robustness fault consists of d parameter values that form an invalid value ($d = 1$) or invalid value combination ($d > 1$). In addition, t_2 other parameter values describe a configuration which is also necessary to activate the fault. In contrast, a t_1-factor fault does not consider any semantic information and just consists of t_1 parameter values that are necessary to activate the fault. Therefore, a (d,t_2)-factor robustness fault is equal to a t_1-factor fault if $d+t_2 = t_1$.

Since the scenarios with unary error-detection conditions (`1-1-1`, `1-2-1`, `1-3-1`, ...) contain at most 4-factor faults in the normal control-flow and (1,3)-factor robustness faults, test suites with 4-wise coverage detect almost all of them.

In that sense, test suites with 5-wise coverage are required for the scenarios with binary error-detection conditions because (2,3)-factor robustness faults must be activated. But even though a 5-wise test suite improves the fault detection effectiveness, the time required for generation as well as the size of the resulting test suite grow strongly. For every scenario, `ROBUSTA` produced a result with fewer test inputs which is at least as effective as (or better than) `IPOG-C` with a strength of five.

This finding can be transferred to lower strengths as well. For instance, `IPOG-C` with a strength of three generates a test suite that detects almost all t_1-factor faults with $t_1 \leqslant 3$ and almost all (d,t_2)-factor robustness faults with $d + t_2 \leqslant 3$. However, `IPOG-C` with a higher strength is required for (d,t_2)-factor robustness faults with $d + t_2 > 3$. Then, `ROBUSTA` produces a result with fewer test inputs which is at least as effective as (or better than) `IPOG-C`.

Therefore, increasing the strength of `IPOG-C` is not always the best option. The results indicate that `IPOG-C` with a strength of t_1 is as effective as `ROBUSTA` as long as the (d,t_2)-factor robustness faults can be considered a special kind of t_1-factor faults, i.e. as long as $d + t_2 \leqslant t_2$ for all (d,t_2)-factor robustness fault. When (d,t_2)-factor robustness faults require a higher strength for `IPOG-C`, i.e. $d+t_2 > t_1$, `ROBUSTA` requires less test inputs and is more effective in terms of fault detection.

F. Threads to Validity

We compared the fault detection effectiveness of an alternative combination strategy by comparing `ROBUSTA` and `IPOG-NEG` to `IPOG-C`. Since the comparison is based on actual test generation and execution, our results might depend on the implementation of the combination strategies as well as the implementation and design of the test scenarios.

To ensure unbiased implementations of the combination strategies, we follow the suggestions for efficient implementations by Kleine and Simos [38].

[5]To distinguish different strengths, we use t_1 for t-factor faults and t_2 for (d,t)-factor robustness faults.

DOI 10.1109/QRS.2019.00018

Table II
FAULT DETECTION EFFECTIVENESS AND TEST SUITE SIZE OF SCENARIOS

Strategy	IPOG-C						ROBUSTA											
Strength	3		4		5		3-2		3-3		4-0		4-1		4-2		4-3	
Scenario	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**
1-1-1	199	0.88	923	0.99	3434	1.00	220	0.93	544	0.95	402	0.74	417	0.86	513	0.98	837	1
1-1-2	199	0.84	923	0.99	3433	1.00	344	0.94	814	0.95	845	0.74	865	0.86	989	0.99	1460	1
1-1-3	199	0.74	923	0.92	3433	0.99	335	0.93	508	0.95	916	0.74	936	0.86	1043	0.98	1216	1
1-1-4	199	0.55	923	0.76	3433	0.90	255	0.93	375	0.95	923	0.74	943	0.86	973	0.98	1092	1
1-2-1	552	0.86	3619	0.99	19428	1.00	603	0.89	1873	0.93	2041	0.69	2066	0.82	2278	0.96	3548	1
1-2-2	552	0.78	3619	0.97	19437	1.00	827	0.88	2326	0.92	3462	0.69	3493	0.82	3742	0.96	5242	1
1-2-3	552	0.63	3619	0.85	19437	0.97	809	0.88	1505	0.92	3617	0.69	3647	0.82	3867	0.97	4563	1
1-2-4	552	0.45	3619	0.69	19440	0.83	676	0.88	1044	0.92	3615	0.69	3645	0.82	3735	0.97	4103	1
1-3-1	324	0.91	1953	1.00	10737	1.00	473	0.96	1493	0.97	1367	0.72	1387	0.84	1580	0.99	2600	1
1-3-2	324	0.88	1949	1.00	10738	1.00	559	0.96	1757	0.96	1950	0.72	1970	0.84	2179	0.99	3378	1
1-3-3	324	0.77	1952	0.96	10740	1.00	547	0.95	1651	0.96	1962	0.72	1982	0.84	2180	0.99	3284	1
1-3-4	324	0.58	1954	0.84	10740	0.94	533	0.93	1515	0.95	1957	0.72	1977	0.84	2162	0.98	3144	1
1-4-1	881	0.90	7572	1.00	58759	1.00	1162	0.93	4379	0.96	5895	0.69	5925	0.81	6308	0.97	9525	1
1-4-2	881	0.82	7553	0.99	58723	1.00	1319	0.93	4839	0.95	7536	0.69	7567	0.81	7969	0.98	11490	1
1-4-3	881	0.67	7571	0.89	58724	0.99	1300	0.92	4579	0.94	7589	0.69	7619	0.81	8004	0.97	11283	1
1-4-4	881	0.50	7565	0.74	58715	0.90	1279	0.92	4272	0.94	7578	0.69	7608	0.81	7971	0.98	10964	1
2-1-1	324	0.85	1949	0.98	10741	1.00	473	0.94	1493	0.95	1369	0.72	1389	0.84	1582	0.99	2602	1
2-1-2	324	0.87	1950	1.00	10747	1.00	559	0.96	1757	0.96	1945	0.72	1965	0.84	2174	0.99	3373	1
2-1-3	324	0.78	1952	0.94	10747	0.99	547	0.94	1651	0.96	1966	0.72	1986	0.84	2184	0.99	3288	1
2-1-4	324	0.58	1949	0.82	10741	0.96	533	0.94	1515	0.96	1956	0.72	1976	0.84	2160	0.98	3143	1
2-2-1	881	0.85	7557	1.00	58725	1.00	1162	0.91	4378	0.94	5899	0.69	5929	0.81	6312	0.97	9528	1
2-2-2	881	0.80	7563	0.98	58728	1.00	1319	0.92	4839	0.93	7543	0.69	7573	0.81	7976	0.98	11497	1
2-2-3	882	0.66	7561	0.89	58768	0.99	1300	0.92	4579	0.94	7587	0.69	7617	0.81	8001	0.97	11280	1
2-2-4	882	0.49	7562	0.73	58715	0.91	1279	0.91	4272	0.93	7561	0.69	7591	0.81	7953	0.98	10946	1

The scenarios used for testing are artificial and do not necessarily represent real-world scenarios. However, we explicitly stated the considered characteristics and measured the implications in the controlled experiment. Therefore, the findings can be used to decide between the two alternatives in real-world scenarios.

To prevent faults being discovered by accident or because of symmetries between the IPM and the parameter values of the SUT, the scenarios are randomized and executed ten times. The presented numbers are average numbers.

To allow and support repeatability of the experiment, the source code of the combination strategies, a set of JUnit5 tests to execute the scenarios as well as the results are published[4].

VI. CONCLUSION & FUTURE WORK

In this paper, we argue that error-handling leads to invalid input masking which requires special treatment in combinatorial testing. We analyze robustness faults and distinguish configuration-independent from -dependent robustness faults and propose a robustness fault model that includes t-factor faults in the normal control-flow and (d,t)-factor robustness faults in exceptional control-flows. The t-wise and single error coverage criteria are adapted accordingly. A test suite that satisfies positive t-wise coverage reaches all t-factor faults in the normal control-flow and a test suite that satisfies negative t-wise coverage reaches all (d,t)-factor robustness faults.

Further, we propose to model invalid values and invalid value combinations directly via error-constraints. The combination strategy `ROBUSTA` describes how to generate test inputs that satisfy positive and negative t-wise coverage by delegating the generation to the existing combination strategy `IPOG-C` and to our modified combination strategy `IPOG-NEG`.

The proposed approach including modelling via error-constraints and generation via `ROBUSTA`, `IPOG-C` and `IPOG-NEG` is implemented as a prototype[4].

As an evaluation, `ROBUSTA` is compared to `IPOG-C` by applying the prototype to 24 artificial scenarios. Test inputs of different strengths are generated ten times for each scenario producing 5040 results. The results indicate that `IPOG-C` with a strength of t is effective in detecting all t_1-factor faults with $t_1 \leqslant t$ and all (d,t_2)-factor robustness faults with $d+t_2 \leqslant t$. In contrast, `ROBUSTA` has the same or higher effectiveness and comparable test suite sizes but requires additional work to model error-constraints.

But, `IPOG-C` with a higher strength is required for (d,t_2)-factor robustness faults with $d + t_2 > t$. Then, `ROBUSTA` is favourable because the test suites are much smaller and provide the same or higher fault detection effectiveness.

In future work, we will evaluate the approach using real-world examples. In addition, we will focus on supporting the modelling of error-constraints because the additional manual work can be an impediment.

REFERENCES

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

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

DOI 10.1109/QRS.2019.00018

[3] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. USA: John Wiley & Sons, Inc., 2005.

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

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

[6] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press, 2007, pp. 255–260.

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

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

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

[10] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *ECOOP'97 — Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 85–103.

[11] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of Systems and Software*, vol. 59, no. 2, pp. 197 – 222, 2001.

[12] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan, "Eh-recommender: Recommending exception handling strategies based on program context," in *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, Dec 2018, pp. 104–114.

[13] J. Lang and D. B. Stewart, "A study of the applicability of existing exception-handling techniques to component-based real-time software technology," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 2, pp. 274–301, Mar. 1998.

[14] Z. Micskei, H. Madeira, A. Avritzer, I. Majzik, M. Vieira, and N. Antunes, *Robustness Testing Techniques and Tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 323–339.

[15] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.

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

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

[18] P. C. Jorgensen, *Software Testing: A Craftsman's Approach, Fourth Edition*, 4th ed. CRC Press, 2014.

[19] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empirical Software Engineering*, vol. 11, no. 4, pp. 583–611, Dec 2006.

[20] R. Tzoref-Brill, "Advances in combinatorial testing," in *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 79 – 134.

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

[22] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, 1997.

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

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

[25] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing faults in exception handling: A study of open source projects," in *2014 Brazilian Symposium on Software Engineering*, Sep. 2014, pp. 11–20.

[26] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: An empirical study of evolving c# programs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 584–595.

[27] D. Sena, R. Coelho, and U. Kulesza, "Integrated analysis of exception flows and handler actions in java libraries: An empirical study," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016, pp. 1520–1526.

[28] J. Oliveira, N. Cacho, D. Borges, T. Silva, and F. Castor, "An exploratory study of exception handling behavior in evolving android and java applications," in *Proceedings of the 30th Brazilian Symposium on Software Engineering*, ser. SBES '16. New York, NY, USA: ACM, 2016, pp. 23–32.

[29] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 413–422.

[30] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, Sep. 2008.

[31] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack," *Empirical Software Engineering*, Jul 2018.

[32] M. Anastasopoules and C. Gacek, "Implementing product line variabilities," in *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, ser. SSR '01. New York, NY, USA: ACM, 2001, pp. 109–117.

[33] H. Melo, R. Coelho, U. Kulesza, and D. Sena, "In-depth characterization of exception flows in software product lines: an empirical study," *Journal of Software Engineering Research and Development*, vol. 1, no. 1, p. 3, Oct 2013.

[34] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, May 1999, pp. 285–294.

[35] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Estimating t-way fault profile evolution during testing," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2. IEEE, 2016, pp. 596–597.

[36] A. W. Williams and R. L. Probert, "A measure for component interaction test coverage," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, ser. AICCSA '01. Washington, DC, USA: IEEE Computer Society, 2001.

[37] 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.

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

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

[40] 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, Sept 2015.

[41] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, 1st ed. Chapman & Hall/CRC, 2013.

[42] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A fault model for subtype inheritance and polymorphism," in *Proceedings 12th International Symposium on Software Reliability Engineering*, Nov 2001, pp. 84–93.