A Case Study on Robustness Fault Characteristics for Combinatorial Testing - Results and Challenges

Konrad Fögen
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
foegen@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, NRW, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Combinatorial testing is a well-known black-box testing approach. Empirical studies suggest the effectiveness of combinatorial coverage criteria. So far, the research focuses on positive test scenarios. But, robustness is an important characteristic of software systems and testing negative scenarios is crucial. Combinatorial strategies are extended to generate invalid test inputs but the effectiveness of negative test scenarios is yet unclear. Therefore, we conduct a case study and analyze 434 failures reported as bugs of an financial enterprise application. As a result, 51 robustness failures are identified including failures triggered by invalid value combinations and failures triggered by interactions of valid and invalid values. Based on the findings, four challenges for combinatorial robustness testing are derived.**

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

I. Introduction

Combinatorial testing (CT) is a black-box approach to reveal conformance faults between the system under test (SUT) and its specification. An input parameter model (IPM) with input parameters and interesting values is derived from the specification. Test inputs are generated where each input parameter has a value assigned. The generation is usually automated and a *combination strategy* defines how values are selected [1].

CT can help detecting interaction failures, e.g. failures triggered by the interaction of two or more specific values. For instance, a bug report analyzed by Wallace and Kuhn [2] describes that "the ventilator could fail when the altitude adjustment feature was set on 0 meters and the total flow volume was set at a delivery rate of less than 2.2 liter per minute". The failure is triggered by the interaction of `altitude=0` and `delivery-rate<2.2`. This is called a *failure-triggering fault interaction* (FTFI) and its dimension is $d = 2$ because the interaction of two input parameter values is required.

Testing each value only once is not sufficient to detect interaction faults and exhaustively testing all interactions among all input parameters is almost never feasible in practice. Therefore, other combinatorial coverage criteria like t-wise where $1 \leq t < n$ denotes the testing strength are proposed [1].

The effectiveness of combinatorial coverage criteria is also researched in empirical studies [2]–[7]. Collected bug reports are analyzed and FTFI dimensions are determined for different types of software [4]. If all failures of a SUT are triggered by an interaction of d or fewer parameter values, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4]. No analyzed failure required an interaction of more than six parameter values to be triggered [7], [8]. The results indicate that 2-wise (pairwise) testing should trigger most failures and 4 to 6-wise testing should trigger all failures of a SUT.

However, so far research focuses on *positive* test scenarios, i.e. test inputs with valid values to test the implemented operations based on their specification. Since robustness is an important characteristic of software systems [9], testing of negative test scenarios is crucial. Invalid test inputs contain invalid values, e.g. a string value when a numerical value is expected, or invalid combinations of otherwise valid values, e.g. a begin date which is *after* the end date.

They are used to check proper error-handling to avoid abnormal behavior and system crashes. Error-handling is usually separated from normal program execution. It is triggered by an invalid value or an invalid value combination and all other values of the test input remain untested. Therefore, a strict separation of valid and invalid test inputs is suggested and combination strategies are extended to support generation of invalid test inputs [1], [10]–[13].

But, the effectiveness of negative test scenarios is unclear as this is not yet empirically researched. To the best of our knowledge, it is only Pan et al. [14], [15] who characterize data of faults from robustness testing. Their results obtained from testing robustness of operating system APIs indicate that most robustness failures are caused by single invalid values. Though, there is no more information on failures caused by invalid value combinations. Because only one type of software is analyzed, more empirical studies are required to confirm (or reject) the distribution and upper limit of FTFIs for other software types (Kuhn and Wallace [4]).

To gather more information on failures triggered by invalid value combinations, we conducted a case study to analyze bug reports of a newly developed distributed enterprise application for financial services. In total, 683 bug reports are examined and 434 of them describe failures which are further analyzed.

The paper is structured as follows. Section II and III summarize foundations and related work. In Section IV, the design of the case study is explained. The results are discussed

$$p_1 : PaymentType \quad V_1 = \{CreditCard, Bill\}$$
$$p_2 : DeliveryType \quad V_2 = \{Standard, Express\}$$
$$p_3 : TotalAmount \quad V_3 = \{1, 500\}$$

Listing 1: Exemplary IPM for a Checkout Service

Table I: Pairwise Test Suite

PaymentType	DeliveryType	TotalAmount
Bill	Express	500
Bill	Standard	1
CreditCard	Standard	500
CreditCard	Express	1

in Section V and challenges for combinatorial testing are discussed in Section VII. Afterwards, potential threads to validity are discussed and we conclude with a summary of our work.

II. Background

A. Robustness Testing

Testing is the activity of stimulating a system under test (SUT) and observing its response [16]. System testing (also called functional testing) is concerned with the behavior of the entire system and usually corresponds to business processes, use cases or user stories [17]. Both, the stimulus and response consist of values. They are called test input and test output, respectively. In this context, input comprises anything explicable that is used to change the observable behaviour of the SUT. Output comprises anything explicable that can be observed after test execution.

A test case covers a certain scenario to check whether the SUT satisfies a particular requirement [18]. It consists of a test input and a test oracle [19]. The test input is necessary to induce the desired behavior. The test oracle provides the expected results which can be observed after test execution if and only if the SUT behaves as intended by its specification. Finally, the expected result to the actual result are compared to determine whether the test passes or fails.

Since robustness is an important software quality [9], testing should not only cover positive but also negative scenarios to evaluate a SUT. Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [18]. Positive scenarios focus on valid intended operations of the SUT using valid test inputs that are within the specified boundaries. Negative scenarios focus on the error-handling using invalid test inputs that are *outside* of the specified boundaries. For instance, input that is malformed, e.g. a string input when numerical input is expected, or input that violates business rules, e.g. a begin date which is *after* the end date.

B. Combinatorial Testing

Combinatorial testing (CT) is a black-box approach to reveal interaction failures, i.e. failures triggered by the interaction of two or more specific values, because the SUT is tested with varying test inputs. A generic test script describes a sequence of steps to exercise the SUT with placeholders (variables) that represent variation points [17]. The variation points can be used to vary different inputs to the system, configuration variables or internal system states [8]. With CT, varying test inputs are created to instantiate the generic test script.

An input parameter model (IPM) is created for which input parameters and interesting values are derived from the specification. The IPM is represented as a set of n input parameters $IPM = \{p_1, ..., p_n\}$ and each input parameter p_i is represented as a non-empty set of values $V_i = \{v_1, ..., v_{m_i}\}$.

Test inputs are composed from the IPM such that every test input contains a value for each input parameter. Formally, a test input is a set of parameter-value pairs for all n distinct parameters and a parameter-value pair (p_i, v_j) denotes a selection of value $v_j \in V_i$ for parameter p_i.

Listing 1 depicts an exemplary IPM to test the checkout service of an e-commerce system with three input parameters and two values for each input parameter. One possible test input for this IPM is `[PaymentType:Bill, DeliveryType:Standard, TotalAmount:1]`. Formally, a test input $\tau = \{(p_{i_1}, v_{j_1}), ..., (p_{i_n}, v_{j_n})\}$ is denoted as a set of pairs. In this paper, we use the aforementioned notation with brackets which is equal to $\tau = \{(p_1, v_2), (p_2, v_1), (p_3, v_1)\}$.

The composition of test inputs is usually automated and a *combination strategy* defines how values are selected [1]. Since testing each value only once is not sufficient to detect interaction faults and exhaustively testing all interactions among all input parameters is almost never feasible in practice, other coverage criteria like t-wise are proposed.

For illustration, Table I depicts a test suite for the e-commerce example that satisfies the pairwise coverage criterion. For more information on the different coverage criteria, please refer to Grindal et al. [1]. To satisfy the coverage criterion, all pairwise value combinations of $PaymentType \times DeliveryType$, $PaymentType \times TotalAmount$ and $DeliveryType \times TotalAmount$ must be included in at least one test input. If the first test input was not executed, pairwise coverage would not be satisfied because the combinations `[PaymentType:Bill, DeliveryType:Express]`,`[PaymentType:Bill, Total Amount:1]`, `[DeliveryType: Express, Total Amount:1]` would be untested.

In comparison to exhaustive testing, fewer test inputs are required to satisfy the other coverage criteria. But as the example illustrates, problems with only one test input might lead to combinations being not covered and failures that are triggered by these combinations remain undetected.

If we suppose that the checkout service requires a total amount of at least 25 dollar, then two test inputs of the example (Table I) with `[TotalAmount:1]` are expected to abort with a message to buy more products. In those cases, the SUT deviates from the normal control-flow and an error-handling procedure is triggered. The value `[TotalAmount:1]` that is responsible for triggering the error-handling is called *invalid value*. If we also suppose that the checkout service rejects payment by bill for total amounts greater than 300 dollar, then `[PaymentType:Bill, TotalAmount:500]` would trigger

error-handling as well. Even though both values are valid, the combination of them denotes an *invalid value combination*.

Valid test inputs do not contain any invalid values and invalid value combinations. In contrast, an invalid test input contains at least one invalid value or invalid value combination. If an invalid test input contains exactly one invalid value or one invalid value combination, it is called a *strong* invalid test input.

Once the SUT evaluates an invalid value or invalid value combination, error-handling is triggered. The normal control-flow is left and all other values and value combinations of the test input remain untested. They are masked by the invalid value or invalid value combination [13]. This phenomenon is called *input masking effect* which we adapt from Yilmaz et al. [20]: "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".

To prevent input masking, a strict separation of valid and invalid test inputs is suggested [1], [10]–[13]. Combination strategies are extended to support t-wise generation of invalid test inputs. Values can be marked as invalid to exclude them from valid test inputs and to include them in invalid test inputs. The invalid value is then combined with all $(t-1)$-wise combinations of valid values. An extension that we proposed also allows to explicitly mark and generate t-wise invalid test inputs based on invalid value combinations [13].

C. Fault Characteristics

According to IEEE [18], an error is "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." It is the result of a mistake made by a human and is manifested as a fault. In turn, a *fault* is statically present in the source code and is the identified or hypothesized cause of a failure. A *failure* is an external behavior of the SUT, i.e. a behavior observable or perceivable by the user, which is incorrect with regards to the specified or expected behavior.

In CT, we assume that the execution path through the SUT is determined by the values and value combination of the test input. If an executed statement contains a fault that causes an observable failure and if a certain value or a certain value combination is required for executing the statement, then the value or value combination is called a *failure-triggering fault interaction* (FTFI). The number of parameters involved in a FTFI is its *dimension* denoted as d with $0 \leq d \leq n$. For instance, if the checkout service contains a fault and accepts a total amount of one but only if express is chosen as the delivery type, then `[TotalAmount:1, DeliveryType:Express]` is a FTFI with a dimension of two.

In general, different types of triggers exist to expose failures [21]. A *trigger* is a set of conditions to expose a failure if the conditions are satisfied . We focus on FTFI, i.e. failures triggered by test input variations, rather than on failures triggered by ordering or timing of stimulation.

In addition, we introduce the following terms for robustness fault characteristics. A failure is a *robustness failure* if the FTFI contains an invalid value or an invalid value combination. Then, the number of parameters that constitute the invalid value or invalid value combination are denoted as the *robustness size*. In case of an invalid value, the robustness size is one.

The extension to support t-wise generation of invalid test inputs is based on the assumption that failures are triggered by an interaction of an invalid value (or invalid value combination) and a $(t-1)$-wise combination of valid values of the other parameters. This is a robustness interaction and its *robustness interaction dimension* can be computed by subtracting the robustness size from the FTFI dimension. There is no robustness interaction if the robustness size and FTFI dimension are equal, i.e. the robustness interaction dimension is zero. For instance, there is no robustness interaction if `[TotalAmount:1]` or `[PaymentType:Bill, Total Amount:500]` trigger failure. In contrast, the robustness interaction dimension of the aforementioned example `[TotalAmount:1, DeliveryType:Express]` is one because the invalid value interacts with one valid value.

III. Related Work

If the highest dimension of parameters involved in FTFIs is known before testing, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4]. However, d cannot be determined for a SUT a-priori because the faults are not known before testing. Hence, the motivation of fault characterization in black-box testing is to empirically derive fault characteristics to guide future test activities.

Existing research on the effectiveness of black-box testing derives the distribution and maximum of d among different types of software based on bug reports. Wallace and Kuhn [2] review 15 years of recall data from medical devices, i.e. software written for embedded systems. Kuhn and Reilly [3] analyze bug reports from two large open-source software projects; namely the Apache web server and the Mozilla web browser. Kuhn and Wallace [4] report findings from analyzing 329 bug reports of a large distributed data management system developed at NASA Goddard Space Flight Center. Bell and Vouk [5] analyze the effectiveness of pairwise testing network-centric software. They derive their fault characteristics from a public database of security flaws and create simulations based on that data. Kuhn and Okum [6] apply combinatorial testing with different strengths to a module of a traffic collision avoidance system which is written in the C programming language. Though, the experiments use manually seeded "realistic" faults rather than a specific bug database. Cotroneo et al. [21] again analyze bug reports from Apache and the MySQL database system and Ratliff et al. [22] report the FTFI dimensions of 242 bug reports from the MySQL database system.

As concluded by Kuhn et al. [7], the studies show that most failures in the investigated domains are triggered by single parameter values and parameter value pairs. Progressively fewer failures are triggered by an interaction of three or more parameter values. In addition to the distribution of FTFIs, a maximum interaction of four to six parameter values is

identified. No reported failure required an interaction of more than six parameter values to be triggered. Thus, pairwise testing should trigger most failures and 4- to 6-wise testing should trigger all failures of a SUT [8].

Several tools include the concept of invalid values to support the generation of invalid test inputs [10]–[12]. An algorithm that we proposed [13] extends the concept to invalid value combinations. In their case study, Wojciak and Tzoref-Brill [23] report on system level combinatorial testing that includes testing of negative scenarios. There, t-wise coverage of negative test inputs is required because error-handling depends on a robustness interaction between invalid and valid values. Another case study by Offut and Alluri [24] reports on the first application of CT for financial calculation engines but robustness is not further discussed.

From an empirical point of view, the effectiveness of negative test scenarios is yet unclear. To the best of our knowledge, it is only Pan et al. [14], [15] who characterize data on faults from robustness testing. The results of testing robustness of operating system APIs indicate that most robustness failures are caused by single invalid values. Though, there is no more information on failures triggered by invalid value combinations. Also, as Kuhn et al. [4] state, more empirical studies are required to confirm (or reject) the distribution and upper limit of FTFIs for other software types. Therefore, we conducted another case study which is described in the subsequent sections.

IV. Case Study Design

A. Research Method

We follow the guidelines for conducting and reporting case study research in software engineering as suggested by Runeson and Höst [25]. As they state, a case study "investigates a contemporary phenomenon within its real life context, especially when the boundaries between phenomenon and context are not clearly evident". Case study research is typically used for exploratory purposes, e.g. seeking new insights and generating hypotheses for new research.

The guidelines suggest to conduct a case study in five steps. First, the objectives are defined and the case study is planned. As a second step, the data collection is prepared before the data is collected in a third step. Afterwards, the collected data is analyzed and finally, the results of the analysis are reported.

B. Research Objective

The overall objective of this case study is to gather information on the effectiveness of combinatorial testing with invalid test inputs, and to compare the obtained results with the ones of other published case studies. For example, the work by Pan et al. [14], [15] indicates that most robustness failures in operating system APIs are triggered by single values rather than value combinations, i.e. a FTFI dimension and robustness size of one. Hence, our aim is to either confirm or reject this indication for enterprise applications. This leads to the following two concrete research objectives:

RO1: Identify the different FTFI dimensions of robustness failures that can be observed in our case study.

The generation of t-wise invalid test inputs is based on the assumption that failures are triggered by a robustness interaction between the invalid value (or invalid value combination) and $(t-1)$-wise combination of the valid values of the other parameters. If the highest robustness interaction dimension is known before testing, then testing all t-wise invalid test inputs of that dimension should be as effective as exhaustive robustness testing.

RO2: Identify different robustness sizes and derive the robustness interaction dimensions that can be observed in our case study.

C. Case and Unit of Analysis

The case is a software development project from an IT service provider for an insurance company. A new system is developed to manage the lifecycle of life insurances. It is based on an off-the-shelf framework which is customized and extended to meet the company's requirements. In total, the new system consists of 2.5 MLOC and an estimated workload of 5000 person days. The core is an inventory sub-system with a central database to store information on customer's life insurance contracts. In addition, complex financial calculation engines and business processes like capturing and creating new customer insurances are implemented. The business processes also integrate with a variety of different already existing systems which are, for instance, responsible to manage information about the contract partners, about claims and damages and to the support insurance agents.

Since life insurance contracts have decade-long lifespans and rely on complex financial models, the correctness of the system is business critical. Mistakes can have severe effects which can even amplify over the long-lasting lifespans and cause enormous damage to the company. Therefore, thorough testing is important.

Even though, the business processes are managed by the new system, they rely on other systems of which each again relies on other systems. This makes it hard to test the system or its parts in isolation. It is also difficult to control the state of the systems and to observe the complete behavior which makes testing even more complicated.

Therefore, most testing is conducted on a system level within an integrated test environment in which all required systems are deployed. The test design is often based on experience and error-guessing. Tests are executed mostly manually because of the low controllability and observability.

D. Data Collection Procedure

To yield the research objectives, the case study relies on archival data from the aforementioned software development project. A project-wide issue management system contains all bug reports from the project start in 2015 to the productive deployment at the beginning of 2018. In general, a bug report is a specifically categorized issue which coexists with other project management- and development-related issues.

For our case study, we analyzed the issue's title, its category, its initial description, additional information in the comment section and its status. Further on, some bug reports are also connected to a central source code management system. If a bug report does not contain sufficient information, the corresponding source code modifications can be analyzed as well.

The issues are filtered to restrict the analysis to only reasonable bug reports. Therefore, issues created automatically by static analysis tools are excluded. Further on, only issues categorized as bug reports whose status is set to *complete* are considered because we expect only them to contain a correct description on how to systematically reproduce the failure.

E. Data Analysis Procedure

Once the bug reports are exported from the issue management system, each bug report is analyzed one at a time. First, it is checked if the bug report describes a failure in the sense that an incorrect behaviour is observable by the user. Otherwise, the bug report is rejected.

Afterwards, the trigger type of the reported failure is determined. The bug report is not further analyzed if no systematically reproducible trigger is found. It is also rejected if the failure is not triggered by a test input variation but rather triggered by unlikely ordering or timing. If a specific value or value combination is identified to trigger the failure, the dimension of the FTFI is determined in the next step.

Then, the bug report is classified as either *positive* or *negative* depending on whether any invalid values or invalid value combinations are contained. If it is classified as negative, i.e. if it is a robustness failure, the *robustness size* of the invalid value or invalid value combination is determined as well.

A robustness size which is lower than the FTFI dimension indicates a robustness interaction between the invalid value (combination) and valid combinations of the other parameters. If possible, the robustness interaction dimension is also extracted from the bug report.

V. Results and Discussion

A. Analyzed Data

In total, 683 bug reports are analyzed. All reported bugs are revealed and fixed during the development phase of the system. Even though filters are applied to export the bug reports, 249 bug reports are classified as *unrelated*, because the issue management system is also used as a communication and task management tool. For instance, problems with configurations of test environments, refactorings or build problems are categorized as bug reports as well.

The remaining 434 bug reports describe failures, they are classified as follows. Eight bug reports do not provide enough information for further analysis and classification. 38 reported bugs require specific timing and ordering of sequences to be triggered. For instance, one sequence to trigger a failure is to search for a customer, open its details, edit the birthday, press cancel and edit the birthday again. Three reported bugs are related to robustness testing. They are triggered by other systems that timeout and do not response to requests. All these bug reports are excluded form further analysis because CT is about varying test inputs rather than varying sequences and timing.

The remaining 388 bug reports describe failures triggered by some test input. A subset of 176 bug reports describes integration failures with other systems where values are not correctly mapped from one data structure to another. They can be triggered by *any* test input. There are so many reported integration issues (45% out of 434 bug reports) because the system consists of several independently developed components which are early and often integrated with the other components and other systems using one of the test environments.

Finally, 212 bug reports are considered to be suitable for CT, which is 49% of all 434 bug reports that describe failures which is 55% of all 388 bug reports that are triggered by some test input To reproduce one of the bugs reported, the test input requires at least one specific value.

B. Observed FTFI Dimensions

The observed FTFI dimensions for the 212 bug reports are depicted in Table II. Most failures are triggered by single parameter values and parameter value pairs and progressively fewer failures are triggered by 3- and 4-wise interactions. In our case, no reported bug requires an interaction of more than 4 parameters in order to trigger the failure.

Table III presents the cumulative percentage of the FTFI dimensions. The last three columns refer to our case study and shows that 76% of all reported failures require 1-wise (each choice) coverage to be reliably triggered. It adds up to 96% when testing with pairwise coverage and 100% are covered when all 4-wise parameter value combinations are used for testing.

To compare our results, the first columns of the table show the results of previous case studies, briefly introduced in the related work section. The numbers and also the average percentage values are taken from Kuhn et al. [7]. The distribution of FTFIs obtained in our case study is not in contradiction to the other cases. However, the distribution is mostly similar to cases [2] and [4]. While there are no obvious similarities with embedded systems for medical devices [2], the large data management system [4] is probably quite similar to our case in terms of requirements and used technologies. Similar to our case, the bug reports are also from a development project whereas the other studies analyze fielded products [7]. For all three cases, most failures are triggered by single parameter values and almost all failures are triggered by the combination of single parameter values and pairwise parameter value combinations. All failures should be triggered by 4-wise parameter value combinations.

So far, only the dimension of failure-triggering fault interactions is considered but differences between positive and negative scenarios are not discussed.

All in all, 51 robustness failures are identified which are classified as follows. 22 failures are caused by *incorrect error-detection* of abnormal situations because conditions to detect

Table II: Observed FTFI Dimensions

d	All	Positive	Negative
1	162	121	41
2	40	31	9
3	6	5	1
4	4	4	-
5	-	-	-
6	-	-	-

Table III: Cumulative Percentage of FTFI Dimensions

	Previous Studies								Our Study		
d	[2]	[3]a	[3]b	[4]	[5]	[21]	[22]	Avg.	All	Pos.	Neg.
1	66	28	41	67	18	9	49	39.7	76	75	80
2	97	76	70	93	62	47	86	75.9	96	94	98
3	99	95	89	98	87	75	97	91.4	98	98	100
4	100	97	96	100	97	97	99	98.0	100	100	
5		99	96		100	100	100	99.0			
6		100	100					100.0			

abnormal situations are either wrong or missing. Consequently, these abnormal situations are not discovered. For instance, bank transfer is accepted as a payment option even though incorrect or no bank account information is provided.

In 19 cases, reported failures are caused by *incorrect error-signaling*. Errors are signaled if an abnormal situation is detected but the error should be handled somewhere else. For instance, a misspelled first name is detected by a user registration service but the error message complaints a misspelled last name.

For three reported failures, the abnormal situation is correctly detected and the error is correctly signaled. However, the system performs *incorrect error-recovery* because the instructions to recover from the abnormal situation contain faults. For instance, the user is asked to correct wrong input, e.g. a misspelled first name. After the input is corrected, the system does not recover and the corrected input cannot be processed.

In seven cases, failures are triggered by the system's runtime environment. For instance, a `NullPointerException` is signaled when the runtime environment detects unexpected and illegal access of `NULL` values. Since developers did not expect `NULL` values, no respective error-handlers are implemented and the processes terminate. These failures denote *incorrect flows* from error-signaling to error-recovery.

Table II depicts the observed FTFI dimensions and their distribution divided into positive and negative test scenarios. As can be seen, the maximum dimension of robustness interaction is three. Compared to positive test scenarios, the negative scenarios discover fewer failures and the FTFI dimensions are also lower. For single parameter values and parameter value pairs, the ratio is 3:1 of valid vs invalid test inputs and no invalid test inputs are identified for higher dimensions.

While these numbers indicate that most failures are triggered by valid test inputs, we emphasize that the test design is based on experience and error-guessing, robustness testing was not in the focus. Hence, the ratio can also result from a general bias towards testing of positive scenarios which is identified in research [26]–[28].

Nevertheless, these findings underpin the results of Pan et al. [14], [15] who observe that most robustness failures in operating systems APIs are triggered by single invalid values. In their study, 82% of robustness failures are triggered by single invalid values. We observe the same ratio in our case.

The bug reports also demonstrate the importance of *strong* invalid test inputs, i.e. test inputs with exactly one invalid value or exactly one invalid value combination. For instance, the component that manages contracting parties ensure data quality by checking that, e.g. the title of a person matches the gender of the first name and that the first name and family name are correct and not confused with each other. However, when using an unknown invalid title, the system responds with a wrong error message saying that the family name was wrong. If an invalid family name was combined with the unknown title, the failure would not have been discovered.

To yield the second research objective, the 10 invalid test inputs with a FTFI dimension greater than one are further analyzed. As a result, two reported bugs that describe failures with robustness interactions are discovered. Even though a combination of two and three specific parameter values is required to trigger the robustness failures, the robustness size is only one and two, respectively.

Furthermore, two reported bugs require an interaction of invalid value (combinations) and a valid value of another parameter. One reported bug is related to the communication between two systems. The response of the second system contains one parameter value with error information to indicate whether the requested operation succeeded or failed. Another parameter provides details about the internal processing of the request and a certain value indicates an internal resolution of the error. In that case, the calling system is expected to handle the error in a different way.

Another reported bug belongs to the storage of details on contracting parties where one contracting party must be responsible for paying the insurance premiums. This responsibility is stored as a role called *contributor*. If direct debit is chosen as the payment method but an invalid bank account, i.e. an invalid IBAN number, is provided, the resulting error message remains even after the invalid IBAN is replaced by a valid IBAN. While the combination `[payment-method:direct-debit, account-number:invalid]` is required as an invalid combination, the bug report states that this phenomenon could only be observed for `[role:contributor]`.

VI. Threads to Validity

The biggest thread to validity is that case studies are difficult to generalize from [25]. Especially, because only one particular type of software of one company is analyzed. The archival data of the case study is only a snapshot and the ground truth, i.e. the set of all failures that can be triggered, is unknown. Hence, the data set can be biased, for instance, towards positive scenarios which has been observed in research [26]–[28]. Since the bug reports result from tests based on experience

an error-guessing, it may apply here as well.

The data can also be biased towards certain fault characteristics. Relevant and reasonable bug reports may be excluded by our filtering because the bug reports are incorrectly categorized. Maybe not all triggered failures are reported. For instance, a developer who finds a fault might just fix it without creating a bug report.

VII. Challenges for Combinatorial Testing

One challenge in combinatorial testing is to find an effective coverage criteria. Based on the aforementioned empirical studies, a recommendation for positive scenarios is to use pairwise coverage to trigger most failures and 4- to 6-wise coverage that should trigger all failures. For the application in practice, one major challenge is to generate test suites of minimal or small size with 4- or 6-wise coverage.

To test negative scenarios, different challenges in combinatorial testing can be observed.

In our case study, four classes of incorrect error-handling are identified. First, incorrect error-detection is caused by conditions which are either too strict or too loose. Second, incorrect error-signaling results in a wrong type of error to be signaled. Third, incorrect recovery of a signaled error is caused by a fault in the appropriate recovery instructions. Fourth, incorrect flow from error-signaling to error-recovery is caused by a signaled error for which no appropriate recovery instructions are implemented.

Challenge 1 - Avoid the Input Masking Effect: Incorrect error-detection that is caused by a too strict condition can be revealed by positive test input that mistakenly triggers error-recovery. But, revealing a condition that is too loose requires invalid test input that mistakenly does not trigger error-recovery. To ensure that too strict and too loose conditions can be detected, the generation of valid and invalid test inputs must be separated and both sets of test input must satisfy separate coverage criteria.

Challenge 2 - Generate Strong Invalid Test Inputs: Another challenge is the generation of strong invalid test inputs such that one invalid value or invalid value combination cannot mask another. Incorrect error-detection and incorrect error-recovery may remain undetected if the signal that would result from an incorrect condition is masked by the computation of another invalid value or invalid value combination.

Challenge 3 - Consider Invalid Value Combinations: Since the error-detection conditions may depend on an arbitrary number of input values, it is not sufficient to only consider invalid values as most combinatorial testing tools do. As our case study and Pan et al. [14], [15] show, 80% of the robustness failures are triggered by invalid values, i.e. a robustness size of one, but also 20% of the robustness failures require invalid value combinations to be triggered. Error-detection with more complex conditions must be tested as well. Invalid value combinations should be excluded when generating positive test inputs but included when generating invalid test inputs [13]. Therefore, appropriate modeling facilities and algorithms that consider invalid value combinations are another challenge.

Challenge 4 - Support Alternative Coverage Criteria: To reveal incorrect handling and incorrect recovery, the SUT must be stimulated by the failure-triggering invalid test input. The majority of analyzed robustness failures does not indicate any robustness interaction between valid values and invalid values or invalid value combinations. Then, the failure is triggered by the invalid value or invalid value combination. To satisfy the coverage criterion, it is sufficient to have a separate test input for each invalid value or invalid value combination.

However, robustness failures where invalid values or invalid value combinations interact with valid values and valid value combinations could also be observed. The failure is triggered by a t-wise interation of one or more valid values and the invalid value or invalid value combination. For instance, suppose the valid `role=contributor` is responsible for selecting the strategy which is used to process the bank data of a customer. If the invalid combination of `[payment-method:direct-debit]` and `[account-number:invalid]` is handled incorrectly by the selected strategy, then the interaction of all three values is required to trigger the failure.

The observed failures of our case study are in line with a case study by Wojciak and Tzorref-Brill [23] who faced error-handling that would be different depending on firmware in control and system configurations. Different configuration options can also be modelled as input parameters, a robustness interaction of configuration options with invalid values and valid value combinations is also reasonable.

Since only low dimensions of robustness interaction are observed, we believe it is unlikely that the generation of 4- to 6-wise test suites is a challenge here as well. Instead, alternative coverage criteria that, for instance, allow a variable strength interaction with some other input parameters can become a relevant to reduce the number of test inputs.

VIII. Conclusion

The effectiveness of negative test scenarios is unclear from an empirical point of view. We conducted a case study to get information on failures triggered by invalid test inputs. The motivation for our and others studies was that if all failures are triggered by an interaction of d or fewer parameter values, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [4].

In our case study we analyzed bug reports which originate from a development project that manages life insurances. In total, 683 bug reports are analyzed. 434 bug reports describe actual failures and 212 of them are failures triggered by a 2-wise or higher interaction of parameter values.

In general, the distribution of FTFI dimensions conforms to the pattern of previous empirical studies. But in contrast to positive test scenarios, fewer robustness failures with lower FTFI dimensions are identified. Overall, the robustness failures are grouped in four classes: incorrect error-detection, incorrect error-signaling, incorrect recovery from a signaled error and incorrect flow from error-signaling to error-recovery. Most robustness failures (80%) are triggered by single invalid values.

The remaining robustness failures require an interaction of two and three input parameter values. Two reported bugs require an interaction of valid values with an invalid value or invalid value combinations to trigger the robustness failure.

Based on the findings of this case study, we derive challenges for combinatorial robustness testing. To ensure that failures do not remain hidden, possible masking should be reduced. Valid and invalid test inputs should be separated and invalid test inputs should be strong, i.e. should only contain one invalid value or invalid value combination.

Further on, it is not sufficient to only consider invalid values as most combinatorial testing tools do. Invalid value combinations should be excluded when generating valid test inputs but considered for invalid test inputs. Therefore, appropriate modeling facilities and algorithms are required.

Since only low robustness interactions are observed, the generation of test inputs with 4- to 6-wise coverage is not that important for negative scenarios. But, the support of variable strength generation for invalid inputs is another challenge.

Most robustness failures do not involve any robustness interaction. But, there are situations where robustness interactions can be observed since different input values, configuration options or internal states are modelled as input parameter values. Depending on expected costs of failure, t-wise testing of invalid test inputs is an option.

In the future, we will work on facilities that support the modelling of invalid value combinations and we will integrate variable strength in a combinatorial algorithm for invalid input generation. To reduce the number of invalid test inputs, we will conduct experiments to investigate the efficiency of different coverage criteria.

REFERENCES

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

[2] D. R. WALLACE and D. R. KUHN, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 08, no. 04, pp. 351–371, 2001.

[3] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, Dec 2002, pp. 91–95.

[4] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, June 2004.

[5] K. Z. Bell and M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software," in *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*. IEEE, 2005, pp. 221–235.

[6] D. R. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *2006 30th Annual IEEE/NASA Software Engineering Workshop*, April 2006, pp. 153–158.

[7] 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.

[8] R. Tzoref-Brill, "Advances in combinatorial testing," ser. Advances in Computers. Elsevier, 2018.

[9] M. M. Hassan, W. Afzal, M. Blom, B. Lindstrom, S. F. Andler, and S. Eldh, "Testability and software robustness: A systematic literature review," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2015.

[10] 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.

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

[12] 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.

[13] 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.

[14] J. Pan, "The dimensionality of failures - a fault model for characterizing software robustness," *Proc. FTCS '99, June*, 1999.

[15] J. Pan, P. Koopman, and D. Siewiorek, "A dimensionality model approach to testing and improving software robustness," in *AUTOTESTCON'99. IEEE Systems Readiness Technology Conference, 1999. IEEE*. IEEE, 1999, pp. 493–501.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, 2015.

[17] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

[18] IEEE, "Ieee standard glossary of software engineering terminology," *IEEE Std*, vol. 610.12-1990, 1990.

[19] 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.

[20] 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.

[21] D. Cotroneo, R. Pietrantuono, S. Russo, and K. Trivedi, "How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation," *Journal of Systems and Software*, vol. 113, pp. 27 – 43, 2016.

[22] Z. B. Ratliff, D. R. Kuhn, R. N. Kacker, Y. Lei, and K. S. Trivedi, "The relationship between software bug type and number of factors involved in failures," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2016, pp. 119–124.

[23] 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.

[24] J. Offutt and C. Alluri, "An industrial study of applying input space partitioning to test financial calculation engines," *Empirical Software Engineering*, vol. 19, no. 3, pp. 558–581, Jun 2014.

[25] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, Dec 2008.

[26] L. M. Leventhal, B. M. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *Human-Computer Interaction*, L. J. Bass, J. Gornostaev, and C. Unger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 210–218.

[27] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why software testing is sometimes ineffective: Two applied studies of positive test strategy." *Journal of Applied Psychology*, vol. 79, no. 1, p. 142, 1994.

[28] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of negative testing on tdd: An industrial experiment," in *Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister and B. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 91–105.