Empirical Software Engineering (2020) 25:2661–2693
https://doi.org/10.1007/s10664-019-09799-2

How does combinatorial testing perform in the real world: an empirical study

Linghuan Hu[1] · W. Eric Wong[1] · D. Richard Kuhn[2] · Raghu N. Kacker[2]

Published online: 20 April 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract
Studies have shown that combinatorial testing (CT) can be effective for detecting faults in software systems. By focusing on the interactions between different factors of a system, CT shows its potential for detecting faults, especially those that can be revealed only by the specific combinations of values of multiple factors (multi-factor faults). However, is CT practical enough to be applied in the industry? Can it be more effective than other industry-favored techniques? Are there any challenges when applying CT in practice? These research questions remain in the context of industrial settings. In this paper, we present an empirical study of CT on five industrial systems with real faults. The details of the input space model (ISM) construction, such as factor identification and value assignment, are included. We compared the faults detected by CT with those detected by the in-house testing teams using other methods, and the results suggest that despite some challenges, CT is an effective technique to detect real faults, especially multi-factor faults, of software systems in industrial settings. Observations and lessons learned are provided to further improve the fault detection effectiveness and overcome various challenges.

Keywords Combinatorial testing · Combinatorial interaction testing · Test case generation · Empirical study

1 Introduction

Detecting faults in software-based systems before they may trigger disruptions, financial loss, injury or loss of life in accidents is one of the main objectives in software testing. A common

Guest Editor: Hélène Waeselynck

✉ W. Eric Wong
ewong@utdallas.edu

[1] Computer Science Department, University of Texas at Dallas, 800 W Campbell Rd, Richardson, TX 75080, USA

[2] National Institute of Standards and Technology, Gaithersburg, MD, USA

misperception is that testing one factor at a time is sufficient for software testing. However, many studies have suggested that the reality might be worse than we thought. Kuhn et al. (Kuhn et al. 2004) analyzed the bug reports of medical devices, modules of POSIX operating systems, Mozilla web browser, Apache server, and two systems used by NASA. They discovered that many faults in a software system, sometimes more than 50 %, require at least two or more factors to be detected. A comprehensive study on recent catastrophic accidents (Wong et al. 2017) also made similar observations that some of them were caused by multiple factors.

Although we could focus on the interactions of multiple factors during testing, detecting software faults, especially multi-factor faults, is still difficult as it is impractical to test exhaustive combinations of all values of all factors. A promising attempt to address this challenge is combinatorial testing (CT), also called combinatorial interaction testing (CIT). Different strategies of CT, such as pair-wise and *t*-wise testing (Grindal et al. 2005), help testers detect multi-factor faults. Algorithms and tools have also been developed to automate the CT test suite generation process and minimize the number of test cases required for applying CT. This further reduces the testing cost and makes CT much easier to apply. However, despite its ease of use, three research questions must be addressed before CT may be widely used in the industry.

RQ1. Can CT be more effective in terms of fault detection when compared to industry-favored techniques, and what factors lead to CT achieving better results?

RQ2. Can CT be less effective in terms of fault detection when compared to industry-favored techniques, and what factors lead to CT producing worse results? Is it possible to avoid poor performance?

RQ3. Are there any challenges when applying CT in industrial settings?

Industry-favored methods tend to be based on prior experience (such as use cases), random testing, and various types of functional testing. Although many studies (Kuhn et al. 2004; Kuhn et al. 2008; Ratliff et al. 2016; Kuhn and Reilly 2002; Cohen et al. 1997; Cohen et al. 1996; Raunak et al. 2017) suggest that CT can be effective in detecting faults, the above research questions are important in the context of industrial settings. To answer these questions, we worked with four companies to evaluate CT on five industrial software systems. The identities of the companies and the systems tested are withheld because of proprietary concerns. The testing was conducted either on-site or remotely under industrial settings to answer the research questions above. This paper discusses the results of these investigations, and our case study results suggest that despite some challenges, CT is an effective technique for detecting faults in software-based systems in industrial settings.

Our primary findings are:

1. CT can help significantly in detecting multi-factor faults.
2. Detailed requirement documents are important for constructing high-quality input space models (ISMs), which are crucial for CT to be effective in detecting faults.
3. When assigning values to factors of an ISM, boundary values and random values of equivalence partitions of the factors can help detect faults that can be triggered by some values of a single factor (single-factor faults).
4. Test execution and output verification should be conducted in parallel to avoid masking effects of CT, which can reduce testing efficiency.

5. If test execution and output verification are not automated, 3-way or higher strength (e.g., 4-way or 5-way) CT is difficult to apply to a complex system under test (SUT) as higher strength CT may require exponentially larger numbers of test cases.

The remainder of the paper is organized as follows: Section 2 presents the background on CT. In Section 3, we explain our experimental design, such as an overview of the entire testing process, the important details of SUT selection, and ISM construction. The details of each case study, including the descriptions of SUT, input model construction, test case generation, test execution, output verification, and comparisons of the faults detected by CT to those detected by the in-house testing teams, are presented in Section 4. Observations and lessons learned are described in Section 5, followed by threats to validity shown in Section 6. Section 7 and Section 8 present the related studies and our conclusions, respectively.

2 Background

As implied by its name, CT – Combinatorial Testing – detects multi-factor faults in a software-based system using combinatorial methods. The overall process of CT is straightforward. First, one needs to identify the SUT and construct a corresponding ISM, which consists of factors and their possible values that might affect the output of the SUT. Second, a CT test case generation algorithm takes the generated ISM as the input to generate a CT test suite T with strength t, which includes all the combinations of the values of any t factors of the ISM. Finally, one executes the test suite T and verifies the outputs.

2.1 System under test and input space model

We use the following example to illustrate the overall process of CT and the notations shown above. Assume we have an online shopping mobile application, and one of its functions is Placing Orders. The goal is to test whether the application can successfully place orders without any faults. In this case, the SUT is the function – Placing Orders. Next, we need to construct the ISM by identifying the factors that might affect the behavior or output of the SUT. These factors can be the total price, shipping address, shipping method, payment method, etc. Once the factors are identified, we assign the possible values to each factor. Table 1 shows a constructed ISM, including the above four factors and their possible values, with F_1, F_2, F_3, and F_4 representing the total price, shipping address, shipping method, and payment method, respectively.

Table 1 The constructed ISM of function – Placing Orders

	F_1 (Total Price)	F_2 (Shipping Address)	F_3 (Shipping Method)	F_4 (Payment Method)
V_1	\$0	Domestic	Same-Day Delivery	Visa
V_2	\$500	International	2-Day Delivery	Mastercard
V_3	\$1000	–	7-Day Delivery	PayPal
V_4	–	–	–	Gift card

2.2 Test case generation and the CT test suite

Once the ISM is constructed, the next step is to generate the CT test suite for a chosen strength t. For a given ISM, a test suite T with strength t indicates that the test cases of T include all the combinations of values of any t factors. In other words, all the combinations of values of any factors taken t at a time can be found in T. A test suite with strength t is called a t-way test suite, and a t-way combination represents a combination of values of t factors. 2-way test suites and 2-way combinations are also called pair-wise test suites and pair-wise combinations, respectively.

In the past two decades, several CT test case generation algorithms and their supporting tools were developed, such as IPOG (Lei et al. 2008) (implemented in tool ACTS (Automated combinatorial testing for software 2018)), Simulated Annealing (Garvin et al. 2011) (implemented in tool CASA (Combinatorial interaction testing portal 2018)). These algorithms take an ISM as the input and automatically generate t-way test suites, the sizes of which are usually smaller than those created manually. For the ISM, as shown in Table 1, it has a total of 53 (= $3 \times 2 + 3 \times 3 + 3 \times 4 + 2 \times 3 + 2 \times 4 + 3 \times 4$) 2-way combinations. Table 2 shows a 2-way test suite of 12 test cases generated by ACTS using the IPOG algorithm.

2.3 Using constraints to remove invalid test cases

If some combinations of values of several factors will not occur in the actual testing, due to some constraints specified by the requirements, we should remove those invalid test cases. For example, if the requirements of function – Placing Orders – do not allow an order with an international shipping address to be placed with the same-day delivery option, and the option of same-day delivery will become unavailable once an international shipping address is entered to the SUT, we will not be able to execute any test cases with this combination. Therefore, such invalid test cases should be excluded to improve testing efficiency. Tools, such as ACTS, allow you to define constraints to automatically exclude them during the test case generation. Note that using ACTS with defined constraints to prevent invalid test cases does not always reduce the size of the generated test suite. In some cases, the size of the generated test suite remains the same or even becomes larger.

Table 2 A 2-way test suite generated by ACTS using the IPOG algorithm

	F_1 (Total Price)	F_2 (Shipping Address)	F_3 (Shipping Method)	F_4 (Payment Method)
t_1	$0	International	2-Day Delivery	Visa
t_2	$500	Domestic	7-Day Delivery	Visa
t_3	$1000	International	Same-Day Delivery	Visa
t_4	$0	Domestic	7-Day Delivery	Mastercard
t_5	$500	International	Same-Day Delivery	Mastercard
t_6	$1000	Domestic	2-Day Delivery	Mastercard
t_7	$0	Domestic	Same-Day Delivery	PayPal
t_8	$500	International	2-Day Delivery	PayPal
t_9	$1000	International	7-Day Delivery	PayPal
t_{10}	$0	Domestic	Same-Day Delivery	Gift card
t_{11}	$500	International	2-Day Delivery	Gift card
t_{12}	$1000	International	7-Day Delivery	Gift card

Table 3 A 2-way test suite generated by ACTS using the IPOG algorithm with constraints

	F_1 (Total Price)	F_2 (Shipping Address)	F_3 (Shipping Method)	F_4 (Payment Method)
t_1	$0	International	2-Day Delivery	Visa
t_2	$500	Domestic	7-Day Delivery	Visa
t_3	$1000	Domestic	Same-Day Delivery	Visa
t_4	$0	International	7-Day Delivery	Mastercard
t_5	$500	Domestic	Same-Day Delivery	Mastercard
t_6	$1000	International	2-Day Delivery	Mastercard
t_7	$0	Domestic	Same-Day Delivery	PayPal
t_8	$500	International	2-Day Delivery	PayPal
t_9	$1000	International	7-Day Delivery	PayPal
t_{10}	$0	Domestic	Same-Day Delivery	Gift card
t_{11}	$500	International	2-Day Delivery	Gift card
t_{12}	$1000	International	7-Day Delivery	Gift card
t_{13}	$1000	Domestic	2-Day Delivery	Visa

Table 3 shows a generated 2-way test suite without the 2-way combination of "International" and "Same-Day Delivery" using the following constraint defined in ACTS.

$$Shipping\ Address = \text{“}International\text{”} => Shipping\ Method! = \text{“}Same\text{–}Day\ Delivery\text{”}$$

3 Experimental design

In this section, we first present an overview of our experimental design and then describe the details of each step.

Figure 1 shows an overview of our experimental design. Before testing a software system, one needs to determine what is going to be tested, which is called SUT. Without careful planning, one can spend excessive testing resources, such as time and manpower,

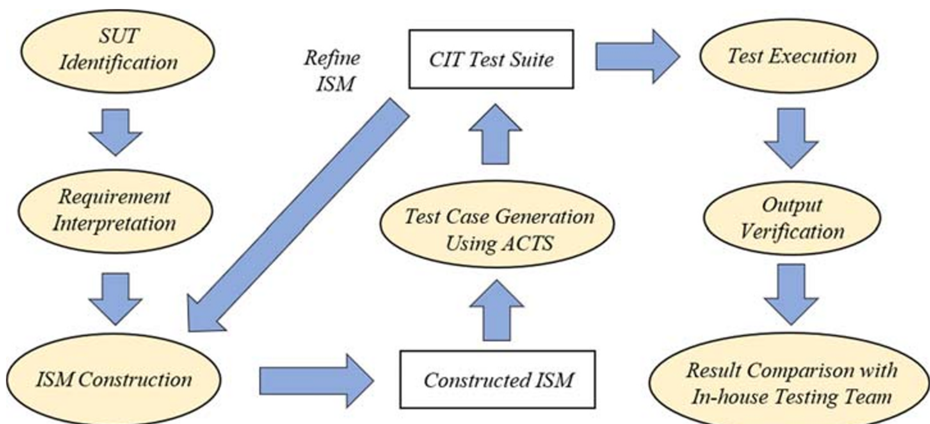

Fig. 1 An overview of our experimental design

but leave a part of the system untested. A common approach is to treat each function of the system as a SUT. We used this strategy in our case studies. For each SUT identified, we interpreted its requirements to construct the corresponding ISM using several techniques, such as equivalence partitioning (Burnstein, Practical software testing 2006), boundary values analysis, and randomization. Once the ISM is constructed, we used ACTS to generate a CT test suite. Depending on the difficulty of the test execution and output verification, we refined our input model several times before we started the actual test execution. This was done to potentially improve the fault detection results, but keep the testing cost to an acceptable level. In our case studies, most of the generated test suites do not have more than 120 test cases for a SUT; this was preferred by all our industrial sponsors. They indicated that any test suite that is larger than this size is likely to require too much time for test execution and output verification. As a result, only 2-way test suites were used in our case studies. The testing was conducted either on-site by us or remotely by the in-house testing teams. Once we finished the test execution and output verification, we compared the faults detected by CT with those detected by the in-house testing team.

3.1 SUT identification

For each industrial system we tested, we first identified the SUTs at function level. By analyzing the requirement documents, we selected those functions whose behavior or output might depend on multiple factors as SUT candidates. This is necessary for applying CT in real-world settings, as CT aims to test the SUT via interactions of multiple factors. If the SUT does not rely on multiple factors in operation, other testing techniques might be more appropriate than CT. For example, in an industrial system we tested, one of the functions is to display demo videos. Based on the specific video chosen in a drop-down menu, the corresponding video will be played on the screen. As a result, there is only one factor, which is the drop-down menu for choosing which video is going to be played. Therefore, this function was not selected as one of our SUTs. Once those candidates were identified, we let the in-house team make the decision of which SUTs we should test.

3.2 Requirement interpretation and ISM construction

For each SUT we selected, we interpreted its requirements provided by the in-house testing team. The requirements of the SUTs identified from the five industrial programs are detailed use case descriptions, user stories, or oral communication with the in-house testing teams. It is important to note that the in-house testing team did not share any testing experiences related to the SUTs (e.g., failure-causing values), since these hints could have affected our ISM construction, which might lead to improved testing results. From a research point of view, in an extreme case, someone with such hints can include all those fault triggering values or combinations in their constructed ISM. As a result, they will most likely outperform whatever methods were previously used by another team, which exaggerates the performance of CT in industrial settings. Although, someone could claim that they will fairly use the information provided by in-house teams in ISM construction. However, the adverse impact on the experimental results can hardly be removed. Using our approach, other testers who apply CT using requirements can achieve similar results, and experienced testers who have more domain-knowledge can achieve even better results.

After we interpreted the requirements, we identified the factors that might affect the behavior or output of the SUT and then assigned possible values for them. For each factor, we used the following two strategies to assign possible values based on its input domain.

3.2.1 Assigning a value as a concrete value

The first strategy is to assign a value as a concrete value. Using this approach, each value of a factor is a concrete value, and it cannot be further divided into sub-values. Regarding the example as shown in Table 1, the values of F_3 are concrete values. This strategy was applied to factors that have small input domains, such as a drop-down menu.

3.2.2 Assigning a value as an abstract value

The second strategy is for factors that have large input domains, such as an integer, a float number, or a string. When using this strategy, we applied the equivalence partitioning technique to create equivalence partitions for each of those factors based on our interpretation of the requirements. This is also suggested by other studies (Kuhn et al. 2008; Kuhn and Okum 2006; Kruse et al. 2013; Reddy 2015). After we obtained the equivalence partitions, instead of selecting one value from each equivalence partition and using them as the values of the factor, we assigned the equivalence partitions themselves to be the values, and we call them the abstract values. For example, the values of the factor F_2, as shown in Table 1, are abstract values. This is because “Domestic” and “International” are two types of shipping addresses, and a specific address is needed in the actual test execution. Once the test cases with abstract values were generated, we replaced each abstract value with a random value from its partition. In this way, we can use random values from partitions to increase the chance of detecting single-factor faults while keeping the generated CT test suite the same size as those generated by selecting one value from each equivalence partition. In addition, we also include boundary values of some identified partitions if we are able to keep the testing cost within the acceptable level.

3.2.3 Refine ISM before test execution

In industrial settings, test execution without automation and output verification are time consuming. We also made the same observations in our case studies. Because of that, we could not conduct test execution and refine the ISM multiple times to obtain better results, and therefore, we only refined the ISM before the test execution. The refinement was conducted based on the estimated testing time of the generated test suite and our testing schedule of each case study. If the size of the generated test suite was too large, which requires too much time to test, we applied multiple strategies to reduce its size, such as converting multiple concrete values to an abstract value, combining multiple factors to one factor, excluding some identified values, etc. If the size of the generated SUT was too small, we included additional values for factors to increase the chance of detecting more faults.

3.3 Test generation, execution, and evaluation

Once we constructed the ISM, we used ACTS with the IPOG algorithm to generate the CT test suite for each SUT. In all our five case studies, we only managed to generate and test 2-way test suites, due to the time constraints. Since automated test execution and output verification were not available, we manually executed the test cases and verified the output based on the

requirements. We compared the faults detected by CT with those detected by the in-house testing teams to evaluate the effectiveness of CT, with respect to the fault detection ability. In addition, we investigated the general causes of those faults. The in-house testing teams helped us determine whether each detected fault was single-factor fault or multi-factor fault.

4 Case studies

In this section, we present the details of each case study, including the descriptions of the SUTs, requirement interpretations, ISM constructions, test generations, test executions, and fault detection results.

4.1 Case study 1

The first case study was conducted on a commercial system (hereafter denoted as S_A) for learning, practicing, and creating Chinese calligraphy. As suggested by the in-house testing team, we focused on a subsystem of S_A called Calligraphy Dictionary, which is used for searching, adding, and editing calligraphy characters via a built-in dictionary. The subsystem contains many components, such as panels, buttons, and drop-down menus, and some components have multiple options. We identified four functions – Add Characters, Delete Characters, Change Font, and Change Character Position – as our SUTs.

4.1.1 SUT – Add characters

The first SUT we tested is Add Characters. The system lets the user add simplified Chinese characters via handwriting recognition. Table 4 shows the use case description obtained from

Table 4 The use case description of SUT – Add Characters

Brief Description
The system lets the user add simplified Chinese characters via handwriting recognition.
Basic Flow
1. The user clicks the *Add Characters* button.
2. The user uses the touch screen to write a character inside the *Handwriting Recognition* field.
3. The system processes the handwriting and displays six candidates.
4. The user selects one character from the six candidates.
5. The system displays the selected character in the *Pending Characters* text field.

Alternative Flows
A. Add Multiple Characters
At any moment, the user can add multiple characters before clicking the *Confirm* button.
B. Delete Characters
At any moment, the user can click the *Backspace* button to delete the last pending character.
C. Rewrite
At any moment, the user can click the *Re-write* button to reset the *Handwriting Recognition* field.
D. Confirm
At any moment, the user can click the *Confirm* button to add characters in the *Pending Characters* text field to the *Character Editing* panel.
E. Invalid Characters
1. Any characters except simplified Chinese characters are invalid. If there are invalid characters, the system should add the valid characters, but not invalid characters to the *Character Editing* panel when a user clicks the *Confirm* button.
2. The system should display an error message about the invalid characters.

the in-house testing team for constructing the ISM. We analyzed the use case description and constructed the ISM as shown in Table 5.

F_1 and its values were identified from the alternative flow – Add Multiple Characters. F_2 and F_3 are for the input characters required by the SUT. We considered both valid characters and invalid characters based on the alternative flow – Invalid Characters. For valid characters – simplified Chinese characters – we created four equivalence partitions – commonly-used, less commonly-used, rare, and variant Chinese characters – based on the character sets defined by the Chinese government. We also created three equivalence partitions for invalid characters – English characters, English symbols, and traditional Chinese characters – for testing invalid characters. We created two factors – F_2 and F_3 – for specifying the input characters, because this could create some combinations between different character sets, which might detect some faults. F_4 and F_5 were identified from the alternative flow – "Rewrite" and "Delete Characters."

Table 6 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. For comparison purposes, based on our testing performance, we estimated that exhaustive testing (1152 test cases) would require 18 h to complete, and this was neither feasible for us nor the in-house testing team. Using only values selected by tester experience, it is unlikely that these additional faults would have been found. CT provides a systematic way of detecting these rare faults, without using fully exhaustive test sets. For this SUT, CT detected three new faults, while the in-house testing detected no fault. Table 7 shows all the faults detected by CT. Note that $Fault_{1\text{–}1}$ and $Fault_{1\text{–}2}$ were detected by the random values generated by the abstract values – rare and variant simplified Chinese characters.

4.1.2 SUT – Delete characters

The SUT – Delete Character – allows the user to delete a selected character from the *Character Editing* panel one at a time. To delete a character, the user first selects a character and then clicks the *Delete* button. Table 8 shows the use case description that we obtained from the in-house testing team for constructing the ISM. We analyzed the use case description and constructed the ISM as shown in Table 9.

Similar to the SUT – Add Character – we created two factors, F_1 and F_2, for testing more combinations of input characters. For the value assignment of F_1 and F_2, we assigned value

Table 5 The constructed ISM of SUT – Add Characters

F_1:	Add character mode
Values:	Add one or multiple characters at a time
F_2:	Input characters I
Values:	Empty, four valid sets of simplified Chinese characters (commonly-used, less commonly-used, rare, and variant Chinese characters), English characters, English symbols, traditional Chinese characters
F_3:	Input characters II
Values:	Empty, four valid set of simplified Chinese characters (commonly-used, less commonly-used, rare, variant Chinese characters), English characters, English symbols, traditional Chinese characters
F_4:	The *Backspace* button
Values:	Do not click, Click once, Click twice
F_5:	The *Rewrite* button
Values:	Do not click, Click once, Click twice

Table 6 No. of test cases and testing cost of SUT – Add Character

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
64	34	3 h	1 h

"Empty" to both F_1 and F_2 so that we can include the "F_1 = empty and F_2 = empty" combination to test the alternative flow – Character Editing Panel is Empty. In addition, unsupported characters (the valid characters that can be added to the system but cannot be appropriately displayed on the screen) were assigned to test alternative flow – Unsupported Characters. F_3 was required by the system, and we must select at least one character in order to delete. Note that the input characters combined by F_1 and F_2 can either be empty, two characters, or four characters. Therefore, we applied additional constraints on F_3 to remove invalid combinations, such as selecting the fourth character when there are only two characters in the system. When there were no characters in the system, we simply ignored F_3. Note that constraints are usually used to set F_3 to be "N/A". However, we noticed that the result of using constraints to set F_3 to be "N/A" can be also achieved by simply ignoring the values of F_3 whenever F_1 and F_2 are empty, which is easier to apply.

Table 10 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. As a result, CT detected two new faults, while the in-house testing team detected no faults. The two new faults detected by CT are shown in Table 11.

4.1.3 SUT – Change font

The Change Font SUT lets the user change the font of a selected character. To change the font of a character, the user first selects a character and then chooses a new font from the candidate fonts.

There are five drop-down menus that filter the candidate fonts by year, style, and three candidate authors. Based on the selected character, these five drop-down menus show different options. The way that the five filters work can be described using the following logical expression:

$$Year \;\&\&\; Style \;\&\&\; \left(Author_1 \parallel Author_2 \parallel Author_3\right)$$

Tables 12 and 13 show the use case description obtained from the in-house team and the created ISM, respectively.

Table 7 The fault detection results of SUT – Add Characters

Fault	Description	CT	In-house	Fault Type
$Fault_{1-1}$	Some rare simplified Chinese characters cannot be found	√	–	Single-factor
$Fault_{1-2}$	Some variant Chinese characters cannot be found	√	–	Single-factor
$Fault_{1-3}$	System does not show error messages for a combination of invalid characters	√	–	Multi-factor

Table 8 The use case description of SUT – Delete Character

Brief Description
The system lets the user delete a character from the *Character Editing* panel one at a time.
Basic Flow
1. The user selects a character from the *Character Editing* panel.
2. The user clicks the *Delete* button.

{Delete Button Clicked}
3. The selected character is deleted from the *Character Editing* panel.
4. The system automatically selects the first character on the right of the deleted character in the *Character Editing* panel. If there are no characters on the right of the deleted character, the first character on the left will be selected. If the *Character Editing* panel is empty, the system does not select any character.

Alternative Flows
A. *Character Editing* Panel is Empty
At **{Delete Button Clicked}**, if the *Character Editing* panel is empty, clicking the *Delete* button does not make any changes.
B. Unsupported Characters
At **{Delete Button Clicked},** if the *Character Editing* panel has unsupported characters, the system deletes all unsupported characters along with the selected character.

The characters already added in the system and the five filters described in the use case description were identified as the six factors of our created ISM. F_1 and its values were identified from the basic flow and the alternative flow – Unsupported Character. F_2 was required by the basic flow, and its values were created based on F_1. As described in the use case description, the available options of the year, style, and author filters change based on the selected character. To simulate the way that filters work and avoid invalid test cases, we created a superset including all the possible options for F_3, F_4, …, F_7 based on the values of F_1 and F_2. Then, we used constraints to ensure that the values of F_3, F_4, …, F_7 are all valid according to the selected values of F_1 and F_2.

Table 14 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. As a result, CT detected four faults, while the in-house testing team detected one fault. Table 15 shows all the faults detected by CT and the in-house testing team.

4.1.4 SUT – Change character position

The last SUT is Change Character Position, which lets the user change the display position of a character, either manually or using the auto-positioning feature. Table 16 and Table 17 show the use case description obtained and the constructed ISM, respectively.

Table 9 The constructed ISM of SUT – Delete Characters

F_1:	Input characters I
Values:	Empty, two supported characters, an unsupported character followed by a supported character, a supported character followed by an unsupported character
F_2:	Input characters II
Values:	Empty, two supported characters, an unsupported character followed by a supported character, a supported character followed by an unsupported character
F_3:	Selected character
Values:	The first, second, third, and fourth character
F_4:	The *Delete* button
Values:	Click once, click twice, click ten times

Table 10 No. of test cases and testing cost of SUT – Delete Characters

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
20	16	3 h	1 h

We applied the same strategy to assign two factors – F_1 and F_2 – for input characters to create more combinations. F_3 is required by this function, and F_4 specifies whether to use the auto-positioning feature after changing the two characters' positions. We would like to emphasize that although F_5 is not specified in the requirements, we included this factor for two reasons. First, we were looking for additional factors to increase the chance of detecting faults, because the test execution for this SUT was not time consuming compared to other SUTs. Second, we completed the testing of SUT – Change Font – before this SUT, which led us to wonder whether the font of the characters can affect the system behavior. Therefore, we included a character with non-default font in our ISM. Surprisingly, this led us to detect a fault that was not detected by the in-house testing team.

Table 18 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. Table 19 shows the faults detection results.

4.2 Case study 2

The second case study was conducted on a software system (hereafter denoted as S_B) running in an embedded device for "Internet of Things" (IoT) application. S_B is used to control and monitor water flow of a pipeline. S_B receives several water-flow related data, such as real-time water flow rate, pipe pressure, and tracking expense command. It then displays them on the screen and returns the water flow warning signal, pipe pressure command, etc., to a pipe controller. Table 20 shows the internal format of the data package.

In the case study of the Chinese Calligraphy system, we tested each SUT separately, since different SUTs required different input data. However, the two identified SUTs of S_B – Display Data and Sending System Response – require the same input data. In addition, the test execution and output verification were time consuming. Therefore, we interpreted the requirements of both SUTs and tested them using one constructed ISM to reduce the testing cost.

Tables 21, 22, and 23 show the use case descriptions of the two SUTs and the constructed ISM, respectively.

The ten factors F_1, F_2, ..., F_{10} were all identified from the use case descriptions of both SUTs. F_1 and F_{10} were identified from the alternative flow – Handle Interference Data. Although the interference data can be any value, we specifically added value "0x55aa" as it is the expected value of the package head. This would test whether the system can correctly handle the data with two continuous valid package heads. F_2, F_4, F_6, F_8, and F_9 were identified

Table 11 The fault detection results of SUT – Delete Characters

Fault	Description	CT	In-house	Fault Type
$Fault_{1–4}$	Unselected character is incorrectly deleted by the system	√	–	Single-factor
$Fault_{1–5}$	All unsupported characters are not deleted as expected	√	–	Multi-factor

Table 12 The use case description of SUT – Change Font

Brief Description
The system lets the user change the font of a character.
Basic Flow
{Start}
1. The user selects a character.
{A character is selected}
2. Based on the selected character, the *Drop-down Menus – Year, Style, Author 1, Author 2,* and *Author 3* display corresponding options.
3. The user changes the selections for each *Drop-down Menu.*
4. The system shows the candidate fonts based on the chosen values of *Drop-down Menus – Year, Style, Author* 1, *Author* 2, and *Author* 3.
5. The user selects a desired font.
6. The system applies the selected font to the chosen character.
Alternative Flows
A. *Character Editing* Panel is Empty
At **{Start}**, if the *Character Editing* panel is empty, *Font Selection Panel* and *Drop-down Menus – Year, Style, Author 1, Author 2,* and *Author 3* should be empty.
B. Unsupported Character
At {**A character is selected**}, if the selected character is an unsupported character, the system should set the *Drop-down Menus – Year, Style, Author* 1, *Author* 2, and *Author* 3 – and the *Font Selection* panel as empty.

from the basic flows. In addition to their expected values, we also assigned the invalid values to test the error handling and the corresponding alternative flows. We assigned random values for F_3, as the requirements do not specify any values. For F_5 and F_7, we identified the equivalence partitions based on the descriptions of the basic flow and the two alternative flows – "Real-time Flow Rate is Out of Range" and "Pipe Pressure is Out of Range." We then added boundary values to both factors. We spent 5 h interpreting the requirements, constructing the ISM, and generating a 2-way test suite, T_1, of 48 test cases.

The test execution and output verification were time consuming; it took us approximately three hours to finish all the tests. During the output verification, we noticed that many system tasks were unexecuted due to the alternative flow – Discard Package. We found that the data packages of 40 test cases were discarded as expected. In seven test cases, the data packages were either discarded inappropriately or the system could not correctly handle the interference

Table 13 The constructed ISM of SUT – Change Font

F_1:	Existing Characters
Values:	Empty, two supported characters, an unsupported character followed by one supported character, two supported characters followed by two unsupported characters
F_2:	Selected Character
Values:	The 1st, 2nd, 3rd, and 4th character
F_3:	Year
Values:	Do not apply and eight additional options
F_4:	Style
Values:	Do not apply and six additional options
F_5:	First Author
Values:	Do not apply and eleven additional options
F_6:	Second Author
Values:	Do not apply and eleven additional options
F_7:	Third Author
Values:	Do not apply and eleven additional options

Table 14 No. of test cases and testing cost of SUT – Change Font

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
127	27	5 h	2 h

data to retrieve the data package. As a result, only one test case was successfully executed for displaying the data and sending the system response.

Since we could not fix these faults and execute the test cases again, we generated additional test cases that can avoid these faults to complete the testing. We revised the ISM by removing the values “0x55aa, ¬0x55aa” of F_1 and F_{10}, and the values “¬0x55aa” of F_2 and F_9. In this way, we generated another 2-way test suite, T_2, of 48 test cases to further test the system. Table 24 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional and random testing; and the time spent in testing. Table 25 shows all the faults detected by CT and the in-house testing team.

4.3 Case study 3

The third case study was conducted on an Extract, Transform, and Load System (ETL) (hereafter denoted as S_C), which was developed and has been used internally by a company to load the sale orders from third-party logistics (3PL) to an enterprise resource planning system (ERP), or send the data from the ERP to update the sale orders on 3PL. We conducted our case study on two SUTs, called “Schedule Inbound Jobs” and “Schedule Outbound Jobs”.

Unlike the first and second case studies that use cases provided as the requirements, we obtained the user story as shown in Table 26 from the in-house testing team.

As shown in Table 27, the three factors F_1, F_2, and F_3 were identified from the user stories and the system interface. The values of F_1 are 32 predefined options listed in a drop-down menu. The values of F_2 include one valid value “03:59:59” and four invalid values. These invalid values were generated by introducing invalid syntax (e.g., “03::22:34”) and invalid values (e.g., “06:xx:40” and “10:365:00”). The values of F_3 were randomly generated. Unlike the factor F_2, we could not assign invalid values for F_3, as the front end of the system automatically corrects the invalid value. As shown in Table 28, we spent one hour on ISM construction and generated a 2-way test suite of 160 test cases. The in-house testing team told us they used a similar number of test cases generated by random, functional, and performance

Table 15 The fault detection results of SUT – Change Font

Fault	Description	CT	In-house	Fault Type
$Fault_{1-6}$	Filters – Year, Style, Authors – show incorrect options for some characters	√	√	Single-factor
$Fault_{1-7}$	When an unsupported character is in either one of the first two positions, the system fails to display an empty font selection panel when it is selected	√	–	Multi-factor
$Fault_{1-8}$	Several combinations of author filters and characters cause missing fonts	√	–	Multi-factor
$Fault_{1-9}$	When the character editing panel is empty, the system fails to display an empty font selection panel	√	–	Multi-factor

Table 16 The use case description of SUT – Change Character Position

Brief Description
The system lets the user change the display position of a character in the *Character Editing* panel.
Basic Flow
1. The user selects a character from the *Character Editing* panel.
2. The user drags the selected character to the new position.
3. The system displays the character at the new position.
Alternative Flows
A. Auto-positioning
At any moment in the basic flow, the user can auto-position the characters in the *Character Editing* panel vertically or horizontally.

testing, but they did not share the exact number. The in-house testing team spent 30 min on test execution and output verification for us. Table 29 shows all the faults detected by CT and the in-house testing team.

Compared to the fault detection results by the in-house testing team, we noticed that three bugs were missed by CT. We worked with the in-house testing team to investigate the causes, and here are our findings. $Fault_{3-2}$ is a fault related to daylight saving time (DST). For a job that was scheduled on a day at 3:00 PM before DST begins, the system is expected to automatically adjust the execution time of this job to 4:00 PM after DST begins. However, the system failed to adjust the schedule time with respect to DST. In order to detect this fault, one needs to consider the DST in their ISM. Since this fault can be detected by validating any scheduled jobs before and after DST begins, this fault is considered as a single-factor fault, and this factor was not included in the ISM. This example illustrates the importance of including fault-triggering factors in the ISM.

$Fault_{3-3}$ also needs an additional factor to be detected, but in this case, it is a multi-factor fault. When a scheduled request is submitted, the system requires a small amount of time (usually a couple of seconds) to finalize the request. As a result, if the moment when the system completes the process is later than the first scheduled execution time, the first execution will be skipped. This fault is difficult to detect, as the process usually takes only a couple of seconds. Since the F_3 does not require "second" as a part of the input value, one could only detect this fault by scheduling a job to be executed at a specific time (e.g., 2:00 PM), and submit this scheduled job in the last couple of seconds of the minute before that time (e.g., 1:59 PM 59 Seconds). Therefore, only considering the boundary value of F_3 is not enough to

Table 17 The constructed ISM of SUT – Change Character Position

F_1:	Input Character I
Values:	Empty, two supported characters, an unsupported character followed by a supported character, a supported character followed by an unsupported character
F_2:	Input Character II
Values:	Empty, two supported characters, an unsupported character followed by a supported character, a supported character followed by an unsupported character
F_3:	Selected character
Values:	The first, second, third, and fourth
F_4:	Auto-positioning
Values:	Do not use, horizontal auto-positioning, and vertical auto-positioning
F_5:	Non-default font character
Values:	The first, second, third, and fourth

Table 18 No. of test cases and testing cost of SUT – Change Character Position

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
27	17	2 h	1 h

detect this fault. One needs to also consider an additional factor – job submission time – which is neither explicitly specified in the requirements, nor the user interface of the system.

$Fault_{3–4}$ is a fault related to race conditions. If two jobs have the same value of F_3, the system is supposed to execute the one that was submitted first. However, if these two jobs were submitted at approximately the same time, the one that was submitted later might be incorrectly executed first. We consider this as a multi-factor fault because one needs to consider the values of F_3 of two test cases and job submission times, as well as the submission sequence of both jobs.

4.4 Case study 4[1]

The fourth case study was conducted on an embedded software system (hereafter denoted as S_D) that is responsible for controlling the dashboard of a locomotive. When the locomotive is running, S_D collects data, such as equalizing reservoir pressure, capacity of air flow, and breaking cylinder pressure from multiple sensors, and controls the physical dashboard to display the corrected data. We treated the whole system as a SUT, since its only task is to control the physical dashboard to display the correct data. In this case study, no written requirements were provided by the in-house testing team. Instead, we interpreted the system and the requirements via oral communication with the in-house testing team.

Similar to but more complex than S_B, shown in Section 4.2, S_D retrieves four data packages that consist of a total of 336 bits, representing 199 input parameters in each test execution. One type of parameter takes multiple bits to represent their integer values in binary format (e.g., two bits "11" represents decimal integer of "3"), and the other types of parameters each take only 1 bit to represent their two states – on and off. Table 30 shows the internal format of some input values.

In this case study, it is not practical to generate an ISM that includes 199 parameters with all possible values. We attempted to overcome this challenge by dividing the testing into four rounds, and we applied CT to one data package in each round. In this way, we generated four ISMs – ISM_a, ISM_b, ISM_c, and ISM_d – by including all the parameters and assigning their values based on the requirements and equivalence partitions. Then we generated four 2-way CT test suites using each ISM, while randomly selecting valid values for the other three ISMs. However, this generated a 2-way test suite of about 400 test cases in each round, which still led us to an unacceptable testing cost. We then further reduced the size of each ISM by combining multiple 1-bit original parameters (level-1 parameters) in a byte. For example, we combined the first eight parameters as shown in Table 30 as a new parameter (level-2 parameter). The level-2 parameter is represented using an eight-digit binary number, and each digit represents the status of the corresponding level-1 parameter. For instance, if the status of first level-1 parameter is *on*, while the rest of the parameters are *off*, the value of the level-2 parameter is (10000000). If a level-1 parameter took more than 1 byte, we did not combine it with other

[1] Some description of case study 4 in this section and case study in section 4.5 appeared in (Li et al. 2016).

Table 19 The fault detection results of SUT – Change Character Position

Fault	Description	CT	In-house	Fault Type
$Fault_{1-10}$	Auto-positioning incorrectly resets the non-default fonts of characters to the default fonts	√	–	Multi-factor

parameters. Table 31 shows a segment of the ISM that has a level-2 parameter F_1 and a level-1 parameter F_2 with three possible values.

Using this approach, we constructed four revised ISMs with respect to four data packages, and each ISM contains 18, 21, 23, and 22 parameters. Table 32 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. Due to the testing agreement with the in-house testing team, we can only show the general information in Table 33 regarding the faults detected by CT and the in-house testing team.

A single-factor fault is missed by CT, not due to the ISM construction nor the value selection, but because of the verification process. When we conducted the testing with respect to each data package, the in-house team suggested that the output verification should be conducted on the outputs (e.g., status lights, values, etc.) related to the corresponding package. Since we did not have such domain-knowledge, which outputs should be verified was instructed by the in-house team. When we test a package, one output should be verified as it is controlled by one input value. However, we were not informed by the in-house team to check this output, which caused us to miss the fault.

4.5 Case study 5

The last case study was conducted on a file manager of a Linux operating system (hereafter denoted as S_E). The in-house testing team gave us two SUTs – File Display Panel and File Search Function. Similar to case study 4, we obtained the requirements via oral communications with the in-house testing team, and there was no written requirement documentation.

4.5.1 SUT – File display panel

The File Display Panel is responsible for displaying files and directories on the hard drive. Based on the requirements and the user interface of the SUT, we constructed the following ISM as shown in Table 34.

Table 20 The internal format of the data package of S_B

Byte	Description	Data Format
0–1	Package head	Unsigned 16 Bits
2	Valve controller ID	Unsigned 8 Bits
3	S_B ID	Unsigned 8 Bits
4–7	Real-time water flow rate	IEEE754–1985 floating-point number
8	Track flow and calculate expense indicator	Unsigned 8 Bits
9–12	Pipe pressure	IEEE754–1985 floating-point number
13–14	CRC checksum	8 Bits CRC
15–16	Package end	Unsigned 16 Bits

Table 21 The use case description of SUT – Display Data

Brief Description
The system receives data from a valve controller and displays them on the interface.
Basic Flow
1. The system receives the data packages from a valve controller.
2. The system retrieves the package based on the package head and package end. The package head and package end should be predefined values – 0x55aa.

{Retrieve and Validate Package}
3. The system displays the real-time water flow rate as a decimal number with two decimal places.
4. The system displays the *Track Flow and Calculate Expense Indicator* as integer 1 if the value of it is 1. Otherwise, the system displays it as integer 0.
5. The system displays the pipe pressure as a decimal number with two decimal places.
6. The system records and maintains a cumulative flow volume (*cfv*) and displays it as a decimal number with two decimal places.
7. The system calculates the cumulative expense (*ce*) using the following equation:

When $cfv \leq 5000$ m^3, $ce = cfv \times 1$ RMB/m^3
When $cfv > 5000$ m^3, $ce = 5000$ RMB + $(cfv - 5000) \times 5$ RMB/m^3

Alternative Flows
A. Real-time Flow Rates is Out of Range
At **{Retrieve and Validate Package}** in the basic flow, if the real-time flow rate is larger (exclusively) than 500.00, the system rounds it off to 500.00. If the real-time flow rate is smaller than (exclusively) 0.00, the system rounds it off to 0.00.
B. Pipe Pressure is Out of Range
At **{Retrieve and Validate Package}** in the basic flow, if the pipe pressure is larger (exclusively) than 10.00, the system rounds it off to 10.00. If the pipe pressure is smaller (exclusively) than 0.00, the system rounds it off to 0.00.
C. Flow Tracking and Expense Calculation Indicator is 0
At **{Retrieve and Validate Package}** in the basic flow, if the *Track Flow and Calculate Expense Indicator* is any value other than 0x01, the system does not record the current real-time flow volume in the system.
D. Discard Package
At **{Retrieve and Validate Package}** in the basic flow, if the S_B ID does not equal 0x20, the system discards the package.
At **{Retrieve and Validate Package}** in the basic flow, if the CRC is not correct, the system discards the package.
At **{Retrieve and Validate Package}** in the basic flow, if there is no valid 17 bytes package with 0x55aa as the package head and the package end, the system does not process the received data.
E. Handle Interference Data
At **{Retrieve and Validate Package}** in the basic flow, the system should correctly retrieve the package data if there are interference data before the package head or after the package end.

Each identified factor is an operation of the SUT that can be performed individually and its values are the available options. We generated a 2-way test suite of 19 test cases. Each test case is executed in an order from F_1, F_2, ..., to F_7. We spent 3 h on ISM construction and 1 h of test execution and output verification. Table 35 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. Table 36 shows the fault detection results of File Display Panel. Due to the testing agreement with the in-house testing team, we are not able to show the detailed descriptions of detected faults.

4.5.2 SUT – File search function

The last SUT is the search function of the file manager system. The file search function helps users search files by different criteria, such as file name, file content, modified data, and file

Table 22 The use case description of SUT – System Response

Brief Description

The system receives the data from a valve controller and sends the warning signal for real-time flow rates and the response command of the pipe pressure.

Basic Flow

1. The system receives the data packages from a valve controller.
2. The system retrieves the package based on the fixed package head and package end values – 0x55aa.

{Retrieve and Validate Package}

3. If the real-time flow rate is larger than (inclusively) 300.00m^3, the system sends the warning signal to the valve controller. Otherwise, the system sends the normal signal to the valve controller.
4. If the pipe pressure is larger than (exclusively) 4.00mpa, the system sends the *reduce pressure* command to the valve controller.
5. If the pipe pressure is less than (exclusively) 3.00mpa, the system sends the *increase pressure* command to the valve controller.
6. If the pipe pressure is larger (inclusively) than 3.00mpa and less than (inclusively) 4.00mpa, the system sends the *do nothing* command to the valve controller.

Alternative Flows

A. Discard Package

At **{Retrieve and Validate Package}** in the basic flow, if the S_B ID does not equal 0×20, the system discards the package.

At **{Retrieve and Validate Package}** in the basic flow, if the CRC is not correct, the system discards the package.

At **{Retrieve and Validate Package}** in the basic flow, if there is no valid 17 bytes package with 0x55aa as the package head and the package end, the system does not process the received data.

B. **Handle Interference Data**

At **{Retrieve and Validate Package}** in the basic flow, the system can correctly retrieve the package data if there are interference data before the package head or after the package end.

size (larger or less than a specific size). We constructed the ISM as shown in Table 37 based on the requirements and the user interface of the SUT.

Table 23 The constructed ISM of S_B

F_1:	Interference data before package head
Values:	Empty, 0x55aa, ¬ 0x55aa
F_2:	Package head
Values:	0x55aa, ¬ 0x55aa
F_3:	Valve controller ID
Values:	Randomly generated 1 Byte Number
F_4:	S_B ID
Values:	0x20, ¬ 0x20
F_5:	Real-time water flow rates
Values:	– 0.005, − 0.004, 299.994, 299.995, 500.004, 500.005
F_6:	Flow tracking and expense calculation indicator
Values:	0x01, ¬ 0x01
F_7:	Pipe pressure
Values:	– 0.005, − 0.004, 2.994, 2.995, 3.995, 4.005, 10.004, 10.005
F_8:	CRC
Values:	Correct CRC, incorrect CRC (randomly generated)
F_9:	Package end
Values:	0x55aa, ¬ 0xx55aa
F_{10}:	Interference data after the package end
Values:	Empty, 0x55aa, ¬ 0x55aa

Note that the value ¬ 0x55aa represents any randomly generated four-digit hexadecimal number, except 0x55aa

Table 24 No. of test cases and testing cost of SUT – System Response and Display Data

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
96 ($T_1 + T_2$)	67	4 h	8 h

The constructed ISM contains ten factors. The values of F_7, F_8, F_9, and F_{10} were assigned to be "Enabled" and "Disabled," as they can only be on and off. We assigned values of F_1, F_2, ..., F_6 based on the existing files on the file system that we selected for testing. We spent two hours in the ISM construction and one hour in the test execution and output verification. Using the constructed ISM shown in Table 37, we generated a 2-way test suite of 17 test cases. Similar to the previous SUT, we can only share the general description of the faults detected. Table 38 shows the size of the test suite generated by us; the size of the test suite generated by the in-house testing team using functional testing; and the time spent in testing. Table 39 shows the summary of all the faults detected by CT and the in-house testing team.

Surprisingly, all the multi-factor faults detected by CT were not detected by the in-house testing team. One fault that was missed by the CT requires an input value derived by expert knowledge to detect. However, such expert knowledge was not provided to us.

Table 25 The fault detection results of S_B

Fault	Description	CT	In-house	Fault Type
$Fault_{2-1}$	The system cannot correctly round the pipe pressure off to 10.00 when it is larger (exclusively) than 10.00	√ (By T_2)	√	Single-factor
$Fault_{2-2}$	The system cannot discard the package when the S_B ID is not 0x20	√ (By T_2)	√	Single-factor
$Fault_{2-3}$	The system cannot retrieve the data package when there is an interference data of 0x55aa before the package head	√ (By T_1)	√	Single-factor
$Fault_{2-4}$	The system cannot handle some interference data before the package head	√ (By T_1)	–	Single-factor
$Fault_{2-5}$	The system incorrectly adds the current flow to the cumulative flow when the "track flow and calculate expense indicator" is 0	√ (By T_2)	√	Single-factor
$Fault_{2-6}$	The system incorrectly calculates the total expense for cumulative flow that is larger than 5000.00m^3	√ (By T_2)	√	Single-factor
$Fault_{2-7}$	The system cannot send "increase pressure" response when the pipe pressure is lower than 3Mpa	√ (By T_1 and T_2)	√	Single-factor
$Fault_{2-8}$	The system cannot send warning signal when the flow rate is larger than 300.00m^3	√ (By T_1 and T_2)	√	Single-factor
$Fault_{2-9}$	The valve ID in the response data is incorrect	√ (By T_1 and T_2)	√	Single-factor
$Fault_{2-10}$	The system can no longer retrieve package after receiving an invalid data package	√ (By T_1)	√	Single-factor
$Fault_{2-11}$	The system fails to give reducing-pressure command when pipe pressure is larger than 4.00	√ (By T_2)	–	Multi-factor
$Fault_{2-12}$	The system incorrectly gives the reducing-pressure command when pressure equals 4.00	√ (By T_2)	–	Multi-factor

Table 26 The user stories of SUTs – Schedule Jobs

User story – Schedule Inbound Jobs
As a user, I want to be able to schedule specific jobs to pipe data from the ERP system to specific 3PL partners, so I can indicate that orders need to be shipped or returned.
User story – Schedule Outbound Jobs
As a user, I want to be able to schedule specific jobs to retrieve data from a specific 3PL partner and push it to the ERP system, so I can update the status of an order (shipment or return).

5 Observations and lessons learned

Through the five case studies, we evaluated how CT performs in industrial settings. In this section, we address the research questions raised in Section 1 based on the testing results, our observations, and lessons learned.

RQ1. Can CT be more effective when compared to industry-favored techniques, and what factors lead to CT achieving better results?

Table 40 shows an overview of the faults detected. The first column represents all the conducted case studies, and the second column shows the total number of single-factor and multi-factor faults detected by CT and the in-house testing teams. The number of single-factor and multi-factor faults detected by CT and the in-house testing teams individually are shown in the third and fourth column, respectively. In the five case studies, CT detected 89.3% single-factor faults and 93.3% multi-factor faults, while the in-house testing teams detected 71.4% single-factor faults and 6.7% multi-factor faults. The results show that although CT does not directly help in detecting single-factor faults, our team delivered better results than the in-house teams. By examining those single-factor faults detected by our team, we discovered that among the 28 single-factor faults detected, 8 faults are detected by random values, and 3 faults are detected by boundary values. For those single-factor faults detected by our team, but not the in-house teams, $Fault_{1\text{–}1}$ and $Fault_{1\text{–}2}$, shown in Table 7, are two examples. These two faults are detected by some random characters of F_2 and F_3 in Table 5, and they could be missed if we did not use random values. The in-house team, however, repeatedly used a small set of characters during the testing, which caused them to miss this fault. Another example is a fault on S_D that can only be detected by a boundary value we used during the testing. Because the in-house team did not use boundary values, they did not detect this fault. This suggests that good detection results with respect to single-factor faults can be achieved by using CT together with random and boundary values of the equivalence partitions identified. This matches our expectation that equivalence partitioning, boundary values analysis, and randomization can be used with CT to achieve a good result of detecting single-factor faults. As noted previously,

Table 27 The constructed ISM of SUTs – Schedule Jobs

F_1:	Jobs
Values:	32 pre-defined options of job drop-down menu
F_2:	Interval
Values:	Empty, “03:59:59”, “06:xx:40”, “10:365:00”, and “03::22:34”
F_3:	Next run time
Values:	A random time in the current day, a random time on the 4th day, a random time on the 7th day

Table 28 No. of test cases and testing cost of SUTs – Schedule Jobs

No. of 2-way test cases	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
160	1 h	30 min

constructing an input model is an intrinsic part of CT, and strong single-factor fault detection capability is a by-product of this test engineering practice. The testing described in these case studies is a good example of this phenomenon – i.e., the input model for CT was more effective than conventional practice. In addition, the results show that CT detects significantly more multi-factor faults (93.3% vs. 6.7%) than the in-house testing teams.

In addition to the number of faults detected, we closely examined the causes of those faults to understand how CT performed better than the in-house testing teams. In general, for the single-factor faults, CT performed better simply by including more fault-triggering values. For the multi-factor faults, the 2-way test suite shows its superiority by including the fault-triggering 2-way combinations. The ability of detecting multi-factor faults is particularly significant when the SUT is safety-critical, as are at least two of the systems we studied (water monitoring and train control).

RQ2. Can CT be less effective when compared to industry-favored techniques, and what factors lead to CT producing worse results? Is it possible to avoid bad performance?

There is no silver bullet in software testing, and CT is surely not an exception. Successful application of CT to achieve good results is important, but lessons learned from the cases that achieve mediocre or even poor results can also be valuable, because we can learn from those lessons to understand the limits of CT. Our results show that CT can sometimes be less effective compared to industry-favored techniques. In case study 3, CT missed three faults that were detected by the in-house testing team. We classified these three missed faults and discovered that one is related to the issue of DST, which should be functional requirement-related, and the other two are related to race conditions and performance issues, which should be non-functional requirements. In addition, by examining the constructed ISM and the root causes of these three faults provided by the in-house testing team, we found that they all required additional factors to be detected, some of which were not documented in the requirements.

On one hand, we would undoubtedly include these factors in our ISM to detect those faults if we obtained the corresponding requirements. However, we also noticed that our testing

Table 29 The fault detection results of S_C

Fault	Description	CT	In-house	Fault Type
$Fault_{3-1}$	The system fails to reject a job request with invalid data	√	–	Multi-factor
$Fault_{3-2}$	The system fails to adjust the execution time of scheduled jobs regarding daylight saving time (DST)	–	√	Single-factor
$Fault_{3-3}$	Jobs can be incorrectly skipped once	–	√	Multi-factor
$Fault_{3-4}$	Race condition can cause two jobs to be executed in incorrect sequence	–	√	Multi-factor

Table 30 The internal format of some input values of a data package of S_D

Bit	Description	Data Format
1	Tube 1	0 – Off, 1 – On
2	Tube 2	0 – Off, 1 – On
3	Cylinder 1	0 – Off, 1 – On
4	Cylinder 2	0 – Off, 1 – On
5	Valve 1	0 – Off, 1 – On
6	Valve 2	0 – Off, 1 – On
7	Warning 1	0 – Off, 1 – On
8	Warning 2	0 – Off, 1 – On
9–24	Air pressure	Integer value from 0 to 999

scope was limited by the concept of CT. Although one can argue that we can include all the factors we could possibly imagine in our ISM, this is not a preferred approach in the industrial setting, since CT tends to generate a large number of test cases. Taking the ISM shown in Table 27 as an example, we included all 33 predefined options for factor F_1, rather than selecting a few of them, as we could not identify the equivalence partitions. This led us to discard some values of F_2 and F_3 to reduce the size of the generated 2-way test suite until it was accepted by the in-house testing team. In this case, using CT influenced the decision not to consider other factors, which caused us to miss those three faults. The in-house testing team, however, performed random, functional, and performance testing from the beginning, and this led them to consider these additional non-functional related factors.

To overcome this challenge, we suggest that one should carefully consider their testing scope before applying CT, as it can affect what factors should be included in the actual testing. In addition, one should be careful about whether to use CT if the size of the generated test suite is barely acceptable. Last but not least, one should be aware that excluding some values of some factors, such as drop-down menus, could lead to missing single-factor faults. If many values should be tested, but including them will lead to an unacceptable testing cost, all-values testing could be considered.

RQ3. *Are there any challenges when applying CT in industrial settings?*

In this section, we will discuss the challenges of applying CT in industrial settings, based on our experiences and the feedback from the in-house testing teams.

The in-house testing teams and our team both agreed that identifying factors to construct the ISM is not difficult. A common approach is to interpret the requirements, and treat inputs required by the SUT as factors of the ISM. However, assigning values to those factors can be challenging. On one hand, we noticed that CT tends to generate test suites beyond an acceptable size for a complicated system, even with the help of the CT algorithm. Five in-house testing teams all indicated that a test suite of about or less than 200 test cases is

Table 31 A segment of the constructed ISM

F_1:	Byte 1
Values:	Eight values: 100000000, 01000000, …, 00000010, 00000001
F_2:	Air pressure
Values:	0, 400, 999

Table 32 No. of test cases and testing cost of S_D

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
296	438	10 h	18 h

preferable for a single SUT in the industrial setting, even with the help of automatic test execution and output verification. To address this issue, one needs to exclude some values of factors or even exclude some factors entirely to reduce the size of the ISM. This brings a challenge of how to conduct ISM reduction without significantly limiting testing effectiveness.

Another challenge is that the common CT testing process – CT test suite generation, test execution, and output verification – might not be agile enough in some situations. Referring to case study 2, shown in Section 4.2, a detected fault ended the test execution before the rest of the system can be executed. This is also called the masking effect in CT testing (Dumlu et al. 2011a). As a result, we spent a large amount of time executing the entire test suite, but only tested a small portion of the system, which was inefficient. What makes this challenging is that we could not foresee this problem since we conducted output verification after the test execution. Therefore, we recommend conducting output verification and the test execution in parallel with CT testing to avoid such an issue.

In addition, multiple in-house testing teams indicated that the CT tool – ACTS – we used in the case studies made test generation very efficient. However, sometimes additional work is needed to convert the abstract values to concrete values and make test cases generated by ACTS to actual test cases that can be executed by the SUT, such as test scripts for common shells. This is challenging because ACTS was designed to be a component in a tool chain for testing. As such, it does not have the feature to convert the abstract values of test cases to concrete values. For example, the value "¬ 0x55aa" of parameter F_2 as shown in Table 23 is an abstract value that represents any randomly generated four-digit hexadecimal number, except 0x55aa. In the actual test execution, one needs to replace "¬ 0x55aa" with a concrete value, such as "0xFFFF." If one wants to use only one value to represent the abstract value, this is relatively easy to achieve using the replace function of a text editor. However, this is more difficult if one wants to choose multiple or random values for the abstract values in order to have a better chance of detecting single-factor faults. Although this challenge can be easily solved with shell scripts or other simple code, some testers did not have programming experience. Without the tool support, one can convert the abstract values of test cases to concrete values only by using additional scripts or in-house tools. Although we did not experience this issue in most cases because most of the test executions were conducted via graphical user interface (GUI), which did not rely on shell scripts, multiple in-house testing

Table 33 The fault detection results of S_D

Fault	CT	In-house	Fault Type
6 Faults	√	√	Single-factor
4 Faults	√	–	Single-factor
1 Fault	–	√	Single-factor
12 Faults	√	–	Multi-factor

Table 34 The constructed ISM of SUT – File Display Panel

F_1:	Sort
Values:	Manual, by name, by size, by type, by time modified, by icon
F_2:	Change location
Values:	Do not change, change
F_3:	Rename files or directories
Values:	Rename some files or directories, do not rename
F_4:	Zoom-in function
Values:	33%, 150%, 400%
F_5:	View
Values:	Icon view, list view, title view
F_6:	Reverse the file display order
Values:	Reverse, do not reverse
F_7:	Compact layout
Values:	Enabled, disabled

teams indicated that this function should be considered in the future so that testing cost can be further reduced. Since we only used ACTS in our case studies, we also investigated other tools that we can evaluate for free in regard to this challenge. To the best of our knowledge, we found no tools that offer the ability to convert abstract values of test cases to concrete values. Only TESTONA (TESTONA 2019) supports shell script generation.

6 Threats to validity

In this study, we aimed to evaluate CT's effectiveness in terms of fault detection in industrial settings. We compared the CT testing results with those of the in-house testing teams. A potential threat to our study validity is that the testing performance achieved by the in-house testing teams selected in our case studies might not reflect the average testing performance in the industry. We alleviated this threat by collaborating with four industrial companies from different fields that have conducted software testing for years.

Another potential threat to validity is that our results might not be extended to other systems. We diminished this threat by selecting different kinds of subject systems, such as GUI and interaction-oriented systems (S_A and S_E), embedded systems (S_B and S_D), and a data processing system (S_C). This allows us to be more confident that the results we obtained from the case studies can be extended to other systems.

When applying CT, the constructed ISM can have a significant impact on the testing effectiveness in terms of fault detection. The ISM constructed by other testers might not include the fault-triggering factors and values. To mitigate this threat, we did not obtain any fault-triggering knowledge from the in-house testing teams, and we constructed the ISMs

Table 35 No. of test cases and testing cost of SUT – File Display Panel

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
19	4	3 h	1

Table 36 The fault detection results of SUT – File Display Panel

Fault	CT	In-house	Fault Type
1 Fault	√	√	Single-factor
3 Faults	√	–	Multi-factor

based on the requirements and the system interface, which other testers can also easily conduct. Using this approach, we are confident that the other testers can construct similar ISMs to those we used in the case studies. In addition, we did not intentionally revise the ISM multiple times based on the testing results to obtain optimal performance, which is far from the industrial setting.

The number of test cases can be a potential threat to validity. Ideally, we want to compare the effectiveness between CT and in-house teams using the same number of test cases. However, this is usually not possible whenever a study is conducted under industrial settings. This is because the number of test cases used by an in-house testing team is fixed before someone uses CT to generate test cases, and it is difficult to match the number of test cases generated by CT to the number of test cases used by the in-house team. We tried to match the number of test cases generated by CT to the number of test cases used by the in-house teams in all studies as much as possible. We believe that, by nature, CT tends to generate many test cases, and we have pointed this out to the in-house teams. The in-house teams told us that as long as the generated test cases detect more faults and they can be executed within the acceptable cost level, they would agree that CT is more effective from a practical point of view. When we presented the results to the in-house teams, they told us that using CT seemed much easier to include the combinations of values of multiple factors that could trigger the faults, since an inherent property of CT is that it provides a complete cover of all t-way

Table 37 The constructed ISM of SUT – File Search Function

F_1:	File name
Values:	Empty, two file names randomly selected from the tested file system, one file name does not exist on the tested file system
F_2:	Content contains
Values:	Empty, two strings existed in the files, one string does not exist in the files.
F_3:	Modified data more than
Values:	N/A, 3 days, 10 days
F_4:	Modified data less than
Values:	N/A, 3 days, 10 days
F_5:	Size larger than
Values:	N/A, 1K Byte, 5K Byte
F_6:	Size less than
Values:	N/A, 1K Byte, 5K Byte
F_7:	File is empty
Values:	Enabled, disabled
F_8:	Search hidden files
Values:	Enabled, disabled
F_9:	Search symbolic link
Values:	Enabled, disabled
F_{10}:	Exclude other file systems
Values:	Enabled, disabled

Table 38 No. of test cases and testing cost of SUT – File Search Function

No. of 2-way test cases	No. of test cases (in-house testing team)	Requirement Interpretation & ISM construction	Test generation, execution, and output verification
17	6	2 h	1 h

combinations of parameter values (up to a specified level), while constructing more tests using conventional methods would rarely produce this level of thoroughness. Furthermore, they were not confident that they could detect those faults (especially those multi-factor faults) detected only by CT by including more test cases using their own methods.

Last but not least, we used a random technique to increase the chance of detecting single-factor faults. Due to the time constraints, we were not able to repeat the test execution multiple times. This can be a potential threat to validity. We will add more manpower to make this possible in the future studies.

7 Related studies

In this section, we provide an overview and discuss other case studies of CT on industrial software systems. Before we present the related study, we show the following four (but not limited to) important aspects for an empirical study on CT. First, only showing how CT can be used to detect faults of a SUT without comparison with other techniques is inadequate. Second, the input space model is crucial for the effectiveness and efficiency of CT. On one hand, missing critical factors in ISMs can leave faults undetected. On the other hand, unnecessary parameters or parameter values can increase the size of the generated test suite, which might lead to an unacceptable testing cost. Therefore, the detailed description of the SUT and the ISM construction should be presented to help testers apply CT to their real-world systems. Third, applying CT in industrial settings can bring unexpected challenges, which are less likely to be faced in a laboratory testing environment. Experiences gained from those industrial challenges are helpful. Fourth, a critical experimental setup of evaluating the effectiveness of CT is whether or not to use seeded faults, which are generated by mutation operators. Andrews et al. (Andrews et al. 2005) suggest that the mutation can be used for experimental purposes only with carefully selected mutation operators. A study by Just et al. (Just et al. 2014) indicates that 17% of real faults are not coupled to any mutation operators, which reveals a potential fundamental limitation of using seeded faults. We believe that using seeded faults generated by mutation operators can indeed provide useful insights. Using real faults, however, is preferred.

Based on the four aspects shown above, we manually summarized the case studies identified and present an overview in Table 41 in chronological order. There are three studies that present

Table 39 The fault detection results of SUT – File Search Function

Fault	CT	In-house	Fault Type
1 Fault	–	√	Single-factor
4 Faults	√	–	Multi-factor

Table 40 An overview of the faults detected

	Total no. of Faults		Detected by CT		Detected by In-house Teams	
S_A	Single-factor	4	4	(100.0%)	1	(25.0%)
	Multi-factor	6	6	(100.0%)	0	(0.0%)
S_B	Single-factor	10	10	(100.0%)	9	(90.0%)
	Multi-factor	2	2	(100.0%)	0	(0.0%)
S_C	Single-factor	1	0	(0.0%)	1	(100.0%)
	Multi-factor	3	1	(33.3%)	2	(66.7%)
S_D	Single-factor	11	10	(90.9%)	7	(63.6%)
	Multi-factor	12	12	(100.0%)	0	(0.0%)
S_E	Single-factor	2	1	(50.0%)	2	(100.0%)
	Multi-factor	7	7	(100.0%)	0	(0%)
Total	Single-factor	28	25	(89.3%)	20	(71.4%)
	Multi-factor	30	28	(93.3%)	2	(6.7%)

details from the four aspects. Puoskari et al. (Puoskari et al. 2013) applied CT to a game, called Habbo Hotel, developed by the company Sulake. Hagar et al. (Hagar et al. 2015) shared their insights and the results of applying CT to real projects in large organizations. However, only limited descriptions of systems under test and bug detection were presented. Rao et al. (Rao et al. 2017) conducted a case study on a high-speed railway track circuit receiver.

Table 41 An overview of the related studies

Study	Description of SUT & ISM Construction	Seeded or Real Faults	Comparison with Non-CT Approaches	Challenges of Applying CT
(Cohen et al. 1996)	✓	Real Faults	–	–
(Dunietz et al. 1997)	✓	–	CT vs. Random	✓
(Huller 2000)	✓	–	CT vs. Quasi-Exhaustive	–
(Smith et al. 2000)	✓	Real Faults	CT vs. All-values	–
(Changhai et al. 2006)	✓	Unknown	CT vs. Random	–
(Kuhn and Okum 2006)	✓	Seeded Faults	–	–
(Krishnan et al. 2007)	✓	Unknown	CT vs. In-house	✓
(Dumlu et al. 2011b)	✓	Unknown	–	–
(Borazjany et al. 2012)	✓	Real Faults	–	–
(Zhang et al. 2012)	✓	Real Faults	–	–
(Ghandehari et al. 2013)	✓	Seeded Faults	CT vs. Random	–
(Borazjany et al. 2013)	✓	Seeded + Real Faults	CT vs. Random	–
(Puoskari et al. 2013)	✓	Real Faults	CT vs. In-house	✓
(Mehta and Philip 2013)	✓	Unknown	–	✓
(Patel et al. 2013)	✓	Real Faults	CT vs. In-house	–
(Wojciak and Tzoref-Brill 2014)	✓	Real Faults	–	✓
(Dhadyalla et al. 2014)	✓	Seeded Faults	CT vs. Functional Testing	✓
(Vilkomir and Amstutz 2014)	✓	Unknown	CT vs. Random	–
(Condori-Fernández et al. 2014)	–	Real Faults	CT vs. In-house	✓
(Kruse et al. 2014)	✓	Seeded Faults	CT vs. In-house	–
(Hagar et al. 2015)	✓	Real Faults	CT vs. In-house	✓
(Petke et al. 2015)	–	Seeded Faults	–	–
(Rao et al. 2017)	✓	Real Faults	CT vs. In-house	✓
(Raunak et al. 2017)	✓	Unknown	CT vs. Random	✓
(Wang et al. 2017)	✓	Real Faults	CT vs. Random	–
(Bures and Ahmed 2017)	✓	Seeded Faults	–	–
(Blue et al. 2018)	✓	Real Faults	–	✓

8 Conclusions and future work

To better evaluate CT in the industrial setting, we conducted five case studies on industrial systems with real faults. In each case study, we presented the details of the ISM construction based on the system requirements, test generation, output verification, and faults detected by CT and the in-house testing teams. Our results suggested that CT can detect more faults, especially multi-factor faults, than the functional and random testing favored by the in-house testing teams. However, we faced some challenges, such as the generation of too many test cases and the masking effect in CT testing, which can cause a negative impact on the testing effectiveness and efficiency. Several insights and strategies are provided to potentially further improve the application of CT in industrial settings and overcome such challenges.

By conducting this study, we have introduced CT to several companies. The in-house testing teams were impressed by its testing performance, and they planned to adopt CT into their testing process. In the future, we plan to extend our study by including more SUTs. Meanwhile, we will also investigate the long-term impact of CT on the five companies that we collaborated with.

Acknowledgements This work was funded by the National Institute of Standards and Technology (Grant No. 60NANB17D322) and US National Science Foundation (Grant No. 1757828 and 1822137). The authors would like to thank their industrial associates for providing the SUTs, the testing environments, and the in-house testing data for this study: Yanzhong Gong and Dr. Jian Wang; Guokai He; Dr. Vidroha Debroy; Chunhui Yang, Dong Li, and Jun Lin. The authors would also like to express their appreciation and gratitude to Dr. Ruizhi Gao and Xuelin Li for helping with data collection.

Compliance with ethical standards

Disclaimer Any mention of commercial products in this paper is for information only; it does not imply recommendation or endorsement by the National Institute of Standards and Technology (NIST).

References

Andrews JH, Briand LC, Labiche Y 2005 May. Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th international conference on Software engineering (pp. 402-411). ACM

"Automated combinatorial testing for software — csrc." https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software. [Online; accessed 12–25-2018]

Blue D, Raz O, Tzoref-Brill R, Wojciak P, Zalmanovici M (2018) "Proactive and pervasive combinatorial testing, " In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pp. 144–152, ACM

Borazjany MN, Yu L, Lei Y, Kacker R, Kuhn R (2012) "Combinatorial testing of acts: A case study," in Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pp. 591–600, IEEE

Borazjany MN, Ghandehari LS, Lei Y, Kacker R, Kuhn R (2013) "An input space modeling methodology for combinatorial testing," in Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp. 372–381, IEEE

Bures M and Ahmed BS (2017) "On the effectiveness of combinatorial interaction testing: A case study," in Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on, pp. 69–76, IEEE

Burnstein, Practical software testing: a process-oriented approach. Springer Science & Business Media, 2006

Changhai N, Baowen X, Ziyuan W, Liang S (2006) "Generating optimal test set for neighbor factors combinatorial testing," in Quality Software, 2006. QSIC 2006. Sixth international conference on, pp. 259–265, IEEE

Cohen M, Dalal SR, Parelius J, Patton GC (1996) The combinatorial design approach to automatic test generation. IEEE Softw 13(5):83–88

Cohen M, Dalal SR, Fredman ML, Patton GC (1997) The aetg system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 23(7):437–444

"Combinatorial interaction testing portal." http://cse.unl.edu/~CTportal/. [Online; accessed 12–25-2018]

Condori-Fernández N, Vos T, Kruse PM, Brosse E, Bagnato A (2014) "Analyzing the applicability of a combinatorial testing tool in an industrial environment," Technical Report UU-CS-2014-008

Dhadyalla G, Kumari N, Snell T (2014) "Combinatorial testing for an automotive hybrid electric vehicle control system: a case study," in Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, pp. 51–57, IEEE

Dumlu E, Yilmaz C, Cohen MB, Porter A (2011a) "Feedback driven adaptive combinatorial testing," Proc Int'l Symp Software Testing and Analysis, pp. 243–253

Dumlu E, Yilmaz C, Cohen MB, Porter A (2011b) "Feedback driven adaptive combinatorial testing," In Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 243–253, ACM

Dunietz S, Ehrlich WK, Szablak B, Mallows CL, Iannino A (1997) "Applying design of experiments to software testing: experience report," in Proceedings of the 19th international conference on Software engineering, pp. 205–215, ACM

Garvin BJ, Cohen MB, Dwyer MB (2011) Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir Softw Eng 16(1):61–102

Ghandehari LSG, Bourazjany MN, Lei Y, Kacker RN, Kuhn DR (2013) "Applying combinatorial testing to the siemens suite," In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 362–371, IEEE

Grindal M, Offutt J, Andler SF (2005) Combination testing strategies: a survey. Softw Test Verif Rel 15(3):167–199

Hagar JD, Wissink TL, Kuhn DR, Kacker RN (2015) "Introducing combinatorial testing in a large organization," Computer, no. 4, pp. 64–72

Huller J (2000) "Reducing time to market with combinatorial design method testing," in Proceedings of the 2000 international council on systems engineering (INCOSE) conference

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R. and Fraser, G., 2014, November. Are mutants a valid substitute for real faults in software testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 654-665). ACM

Krishnan R, Krishna SM, Nandhan PS (2007) Combinatorial testing: learnings from our experience. ACM SIGSOFT Softw Eng Notes 32(3):1–8

Kruse PM, Prasetya IWB, Hage J, Elyasov A (2013) "Loggingtofacilitatecombinatorial system testing," In International Workshop on Future Internet Testing, pp. 48–58, Springer

Kruse PM, Shehory O, Tron DC, Fernandez NC, Vos T, Mendelson B (2014) "Assessing the applicability of a combinatorial testing tool within an industrial environment," in 11th workshop on experimental software engineering (ESELAW 2014)

Kuhn R and Okum V (2006) "Pseudo-exhaustive testing for software," in Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA, pp. 153–158, IEEE

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

Kuhn R, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 30(6):418–421

Kuhn R, Lei Y, Kacker R (2008) "Practical combinatorial testing: Beyond pairwise," It Professional, no. 3, pp. 19–23

Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J (2008) Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. Softw Test Verif Rel 18(3):125–148

Li X, Gao R, Wong WE, Yang C, Li D (2016) "Applying Combinatorial Testing in Industrial Settings," In Proceedings of 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 53–60

Mehta M and Philip R (2013) "Applications of combinatorial testing methods for breakthrough results in software testing," in Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp. 348–351, IEEE

Patel S, Gupta P, Shah V (2013) "Combinatorial interaction testing with multi-perspective feature models," in Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp. 321–330, IEEE

Petke J, Cohen MB, Harman M, Yoo S (2015) Practical combinatorial interaction testing: empirical findings on efficiency and early fault detection. IEEE Trans Softw Eng 41(9):901–924

Puoskari T Vos E, Condori-Fernandez N, Kruse PM (2013) "Evaluating applicability of combinatorial testing in an industrial environment: A case study," in Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation, pp. 7–12, ACM

Rao J, Guo N, Li Y, Lei Y, Zhang Y Li Y, Cao Y (2017) "Applying combinatorial testing to high-speed railway track circuit receiver," in Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on, pp. 199–207, IEEE

Ratliff ZB, Kuhn DR, Kacker RN, Lei Y, Trivedi KS (2016) "The relationship between software bug type and number of factors involved in failures," in Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on, pp. 119–124, IEEE

Raunak MS, Kuhn DR, Kacker R (2017) "Combinatorial testing of full text search in web applications". In 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 100–107, IEEE

Reddy M (2015) "Combinatorial testing: A case study approach for software evaluation," In Electrical, Computer and Communication Technologies (ICECCT), 2015 IEEE In- ternational Conference on, pp. 1–5, IEEE

Smith D, Feather MS, Muscettola N (2000) "Challenges and methods in testing the remote agent planner," In AIPS, pp. 254–263

"TESTONA - Assystem Germany GmbH" – https://www.assystem-germany.com/en/products/testona/, [Online; accessed 02–22- 2019]

Vilkomir S and Amstutz B (2014) "Using combinatorial approaches for testing mobile applications," in Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, pp. 78–83, IEEE

Wang S, Wu T, Yao Y, Jin B, Ding L (2017) "Combinatorial testing on mp3 for audio players," in Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on, pp. 272–275, IEEE

Wojciak P and Tzoref-Brill R (2014) "System level combinatorial testing in practice–the concurrent maintenance case study," In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST), pp. 103–112, IEEE

Wong WE, Li X, Laplante PA (2017) Be more familiar with our enemies and pave the way forward: a review of the roles bugs played in software failures. J Syst Softw 133:68–94

Zhang Z, Liu X, Zhang J (2012) "Combinatorial testing on id3v2 tags of mp3 files," in Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 587–590, IEEE

Linghuan Hu is a Ph.D. student in software engineering at the University of Texas at Dallas. His research interests include combinatorial testing and test generation. Hu received a M.S. in software engineering from the University of Texas at Dallas. Contact him at linghuan.hu@utdallas.edu.

W. Eric Wong is a professor and the director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science at the University of Texas at Dallas. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. Wong received a Ph.D. in computer science from Purdue University, West Lafayette, Indiana. He is the Editor-in-chief of IEEE Transactions on Reliability and the corresponding author for this article. Contact him at ewong@utdallas.edu.

Rick Kuhn is a computer scientist in the Computer Security Division of the National Institute of Standards and Technology and is an IEEE Fellow. He has authored three books and more than 150 papers on information security, empirical studies of software failure, and combinatorial methods in software testing, and co-developed the role-based access control model. His current technical interests are in applications of combinatorial methods for software assurance and explainable AI. He received an MS in computer science from the University of Maryland College Park.

Raghu Kacker is a researcher in the National Institute of Standards and Technology. His research interests include development and use of combinatorial methods for testing software-based systems. He has coauthored over 180 papers and one book. He has a Ph.D. and has worked in academia (Virginia Tech), industrial (AT&T Bell Laboratories) and government (NIST) research laboratories. He is a Fellow of the American Statistical Association and a Fellow of the American Society for Quality. He has received the Distinguished Technical Staff Award from AT&T Bell Labs, and Bronze medal and Silver medal from the US Department of Commerce.