Automatic Detection of Parameter Shielding for Test Case Generation*

Jingjian Lin[1,2], Jun Yan[1], and Jifeng Xuan[3,4]

[1]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, China
[2]University of Chinese Academy of Sciences, China
[3]State Key Laboratory of Software Engineering, Wuhan University, China
[4]INRIA Lille - Nord Europe, France
Email: {linjingjian12,yanjun}@otcaix.iscas.ac.cn, jifeng.xuan@inria.fr

Abstract

Parameter shielding refers to the situation that one test parameter disables others in test execution. The quality of test case generation techniques is limited by the wide existence of parameter shielding. It is challenging to automatically find out conditions that cause the parameter shielding. This paper presents a novel approach for exploring the shielding conditions of test parameters. Our approach executes test inputs and collects runtime information of execution as features of test inputs. Then, a clustering algorithm is used to group test inputs with similar runtime information while a decision tree algorithm is built to extract the conditions in the groups. Finally, our approach identifies the shielding conditions based on the decision tree. Experiments on seven programs show that our approach can effectively detect the parameter shielding and the related conditions.

Keywords: *black-box testing, parameter shielding, clustering, decision tree*

1. Introduction

Software companies employ testing techniques as one indispensable step for quality assurance. A Software Under Test (SUT) has multiple input parameters and each parameter may lead to a large input space. In practice, it is expensive to verify the correctness of SUT using exhaustive testing [1], [2]. A variety of test case generation techniques have been proposed to reduce the scale of test cases, such as equivalence partitioning, boundary-value analysis, and category-partition methods [3].

In test case generation for a SUT with more than one parameter, one parameter may be shielded by others. *Parameter shielding* refers to the situation that one parameter disables other parameters in the SUT [4]. For example, if an application opens a modal child window, all operations to the parent window will be shielded; for many command-line tools in Linux system, the parameter `--help` shields all the other parameters. Parameter shielding results in the redundancy and low quality for test cases generation. For example, in combinatorial testing with Mixed Covering Array (MCA), parameter shielding will make test case generation fail in exposing potential errors, which should be detected if no parameter shielding exists [4]. Existing work by Chen et al. [4] focuses on how to generate combinatorial test cases under the scenario of shielding conditions. However, to the best of our knowledge, how to automatically detect the parameter shielding has been unexplored yet.

We propose an approach to automatic parameter shielding detection in this paper. This approach generates test inputs and extracts function call information via dynamic analysis tools. Cluster analysis groups inputs that show the similar function calls while a decision tree algorithm identifies constraints of parameters in clusters. By analyzing the result of the decision tree, we find out whether there exists parameter shielding and extract the condition that causes the parameter shielding. Experiments show that our approach can effectively detect the parameter shielding and the related conditions.

2. Background

2.1. Parameter shielding

Parameter shielding widely occurs in the case that several parameters control the same or relevant program logics [4]. However, in automatic testing, it is hard to be aware of

*Corresponding author: Jifeng Xuan. This work is supported by National Natural Science Foundation of China (under grant No. 91118007) and INRIA Postdoctoral Research Fellowship.

(DOI Reference Number: 10.18293/SEKE2015-039)

Table 1: 2-way test cases of the example

test index	a	b	c
t1	1	1	1
t2	1	2	2
t3	2	1	2
t4	2	2	1

Table 2: 2-way test cases under parameter shielding

test index	a	b	c
t1	1	1	1
t2	1	2	2
t3	2	1	#
t4	2	2	#

parameter shielding before test case generation. This may fail to satisfy the requirements of a specific test case generation technique. Consider the following scenario of combinatorial testing, a SUT has three parameters `a`, `b`, and `c` with valid values of {1,2}. Table 1 shows the test cases generated by 2-way testing. The technique, *t*-way testing, aims to cover every possible combination value of no more than *t* parameters [4], [1].

Assume that parameter `c` is shielded by `a`: when `a==2`, `c` is disabled. Then test cases in Table 1 can be transformed into cases in Table 2 ('#' means the parameter is disabled). We find that test cases in Table 2 have not covered the following value pairs of `b` and `c`: $\langle 2,1\rangle$ and $\langle 1,2\rangle$, which are originally covered in Table 1. In other words, when parameter `a` shields `c`, test cases cannot meet the requirements of 2-way testing. Therefore, it is important to find out the potential parameter shielding before test case generation.

2.2. Data mining

Data mining, aims to extract implicit, previously unknown, and potentially useful information from databases [5]. It is actually the process of finding the hidden data pattern of the databases.

In this paper, we leverage clustering and classification techniques to analyze runtime logs. The goal of clustering is to discover similarities and differences among data patterns in order to derive useful conclusions about similar clusters [6]. According to a specific similarity measure, a data set is divided into clusters; such division ensures that data inside one cluster have a higher similarity than those in different clusters [7].

The goal of classification is to predict categorical labels, such as "safe" or "risky" for the loan application data, "yes" or "no" for marketing data [7]. Decision trees are a typical family of classifiers on a target class in the form of a tree structure. One main advantage of decision trees is to produce a set of rules, which represents the branches and nodes of the tree; such rules can be easily interpreted into condition combinations, comparing with other basic classification techniques [8].

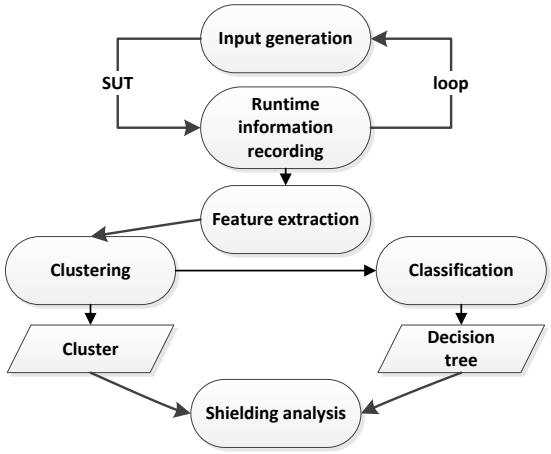

Figure 1: Process of parameter shielding detection

3. Approach

We propose an approach to automatically detecting parameter shielding. Figure 1 shows the process of our approach, which consists of six steps.

3.1 Input generation

In general, parameter shielding occurs when a parameter value dominates a module or several parameters are sensitive to the parameter orders. In our work, to detect parameter shielding, we extract parameters in the same module and generate test inputs for further test execution.

First, we generate test inputs, i.e., input vectors of actual values for parameters, for every single parameter. We can generate all values in the input space for parameters with small input space. For parameters with large input space, we choose parameter values that generate different runtime behaviors, such that these values can be divided into different clusters by clustering analysis. Values can be selected by equivalence partitioning, boundary-value analysis etc. Based on these selected values, we calculate all combination and permutations of parameters. So the number of generated test inputs is $n! \times \prod_{i=1}^{n} v_i$, where n denotes the number of parameters and v_i denotes the number of values for the ith parameter.

3.2. Runtime recording and feature extraction

We execute all generated inputs and extract their runtime information. Using dynamic analysis tools such as Valgrind [9],[1] we are able to collect running information of SUTs, including the orders of function calls and the number of function calls. Then, function call information is stored in logs for feature extraction.

[1] http://valgrind.org/

We describe the two kinds of features in our experiments as follows. The *order of function calls* refers to the index of a function such as `bar` in the call chain of another function such as `foo` during the test execution. For example, if `foo` calls 10 functions in one execution and `bar` is the second function called by `foo`, then the order of `bar` called by `foo` is 2. Similarly, the number of function calls refers to the count of a function such as `bar` called by `foo` during execution. For example, if `foo` calls `bar` nine times, then number of function calls of `bar` called by `foo` is 9.

```
def gen_func_call_pair(logs):
  call_dict = dict()
  for func in calling functions:
    for subfunc in functions called by func:
      if not call_dict[func]:
        call_dict[func] = set()
      call_dict[func].add(subfunc)
  return call_dict
def gen_numeric(call_dict, log):
  features = []
  for func in call_dict:
    for subfunc in call_dict[func]:
      if subfunc is called by func in log:
        feature func:subfunc = its calling order
      else:
        feature func:subfunc = -1
      features.append(func:subfunc)
  return features
def gen_featMat(logs):
    call_dict = gen_func_call_pair(logs)
    for log in logs:
        features = gen_numeric(call_dict, log)
        print features to file
```

Listing 1: Feature extraction of the orders of function calls

The order of function calls can be converted to numeric features by the python-like pseudo code in Listing 1. Function `gen_func_call_pair` scans logs that are record by dynamic analysis tools and finds out the collection of functions which called by the same function. Then we can obtain all function call pairs. For example, `funcA:funcB` means the order of `funcB` in the collection of functions which are called by `funcA`. Function `gen_numeric` is used for generating the number value of specific features. It scans logs that record calling information of a specific input and calculates the feature values. Note that some call pairs only occurs in some specific inputs. Thus, we assign a specific value such as -1 to values of call pairs which are not occurred. Function `gen_featMat` calls `gen_func_call_pair` and `gen_numeric` to generate all values of features and write them to file. The feature extraction for the number of function calls is similar to the feature extraction for the order of function calls.

3.3. Clustering and decision tree algorithms

Based on the extracted features, we apply clustering algorithms based on numeric distances to detect similar test inputs [7]. Clustering, such as k-means and EM algorithms [6], is used in our approach for grouping inputs which generate similar program behaviors together in a cluster. In our work, the k-means algorithm is used for clustering in the implementation. Since we select values that conduct different program behaviors, the inputs will be grouped into different clusters. Recall the example in Table 1, assume that all parameter values lead to different program behaviors. Then case $\langle 2, 1, 1\rangle$ and $\langle 2, 1, 2\rangle$ will be grouped into different cluster. If `c` is disabled when `a==2`, then case $\langle 2, 1, 1\rangle$ and $\langle 2, 1, 2\rangle$ will be grouped in the same cluster, called `cluster1`. This cluster (`cluster1`) is determined by value of `a` and `b` only, meanwhile other clusters are determined by values of all parameters. We can find out the shielding condition by analyzing the clustering result.

To further "understand" clusters, we employ a decision tree algorithm to find out the conditions for parameter shielding. Given clusters as well as their test inputs, we treat a cluster, which a test input belongs to, as the label of the test input. Then we have a data set of labeled test inputs. Based on this data set, we train a decision tree model to build the relationship between features and their labels (clusters). Decision trees are a typical kind of classification algorithms, for example, ID3, C4.5, and CART [10]. C4.5 is used in our implementation.

3.4. Shielding analysis

When a parameter is shielded by others, the parameter value will not affect the program behavior. In other words, when a parameter is disabled, runtime behaviors of the SUT are only determined by other parameters.

We can find the shielded parameter from decision tree by the following steps: first, in a decision tree, we merge paths (conditions) that come from the same cluster; second, we find out parameters that do not exist in the conditions of clusters. These parameters are the shielded parameters while the identified conditions could be the shielding conditions.

4. Experiments

Experiments are conducted on programs of different types and scales. Table 3 lists seven programs in our experiments. The experiments are implemented with open-source data mining platforms, Weka.[2]

4.1 Program with enumeration inputs

Taking program `ln` as an example, three parameters `-s`, `-P`, and `-L` are considered in the experiment. Parameter `-s` is used for making symbolic links instead of hard links; parameter `-P` is used for changing hard links directly into symbolic links; and parameter `-L` is for changing hard links into symbolic link references.

[2] http://www.cs.waikato.ac.nz/ml/weka/

Table 3: Seven SUTs in experiments

	Name	#Parameter	#Feature	#Input	Kloc	Description
1	ls	3	366	12	5.0	print a list of the current directory
2	cp	3	318	4	1.2	copy files or directories
3	ln	3	56	12	0.6	establish link to a file or directory
4	head	2	82	800	1.0	display the first few lines of a file
5	tail	2	108	800	2.3	display the last few lines of a file
6	bzip2	2	182	4	7.3	a compression program
7	ffmpeg	2	6358	200	892.5	solution to audio and video processing

Table 4 shows the classification result of `ln`. We can find out that test inputs of `-L` and `-PL` belong to the same cluster (`cluster1`), while inputs of `-P` and `-LP` belongs to another cluster (`cluster2`). We can infer that when `-P` is in front of `-L`, program behaviors are almost the same as input `-L`. Thus, we conclude that `-P` is shielded if `-P` is listed in front of `-L`. Similarly, `-L` is shielded when `-L` is in front of `-P`. Meanwhile, all inputs that contain the parameter `-s` belong to the same cluster (`cluster0`), so we can conclude that `-P` and `-L` are shielded when `-s` exists in the inputs. We confirm the above conclusions by manually verification.

4.2 Program with integer inputs

We discuss shielding of integer inputs in this subsection. Figure 2 shows the decision tree of parameter `-vframes` $\langle number \rangle$ and `-r` $\langle fps \rangle$ of `ffmpeg`. Parameter `-r` set the number of video frames to output and parameter `-vframes` set frame rate of the input video. Since the path of `cluster0` is not divided by `vframes`, we can conclude that when $r \leq 0$, `vframes` is shielded. By manually executing the program, we confirm that the conclusion is correct.

Table 4: Classification result of `ln` with parameters -L, -P, and -s

cluster0	cluster1	cluster2
-s,-Ls,-sL,-Ps,-sP	-L	-P
-LPs,-LsP,-PLs	-PL	-LP
-PsL,-sPL,-sLP		

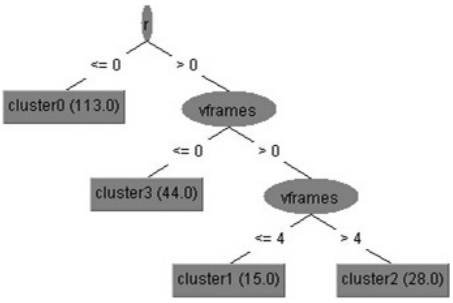

Figure 2: Classification result of `ffmpeg`

Table 5: Classification results of `cp`, `head`, and `bzip2`

Name	Parameter	Cluster	Condition
ls	-g,-A	$cluster0 : -g, -gA, -Ag$ $cluster1 : -A$	-g shields -A
cp	-s,-L	$cluster0 : -s, -sL, -Ls$ $cluster1 : -L$	-s shields -L
head/tail	-n$\langle lines \rangle$ -c$\langle bytes \rangle$	$cluster0 : c \leq 0$ $cluster1 : 0 < c \leq 20$ $cluster2 : c > 20$	-c shields -n
	-c $\langle bytes \rangle$ -n$\langle lines \rangle$	$cluster0 : n \leq 0$ $cluster1 : 0 < n \leq 9$ $cluster2 : n > 9$	-n shields -c
bzip2	-t,-d	$cluster0 : -t, -dt$ $cluster1 : -d, -td$	later parameter shields the former

4.3 Results for other programs

Table 5 shows other results of SUTs in Table 3. For all these programs, we find that the shielding conditions are correctly detected.

5. Conclusion

This paper proposes a novel approach to automatic detection of parameter shielding for test case generation.Test parameters shielded by others are found by clustering the runtime information of the SUT. Experiments show that our approach can effectively detect the parameter shielding for various types of parameters. Meanwhile, shielding conditions between parameters can also be detected in our approach. Our shielding detection approach can be used for enhancing the quality of test cases and for reducing the testing cost.

References

[1] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*. Springer, 2014.

[2] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 52–63, ACM, 2014.

[3] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[4] B. Chen, J. Yan, and J. Zhang, "Combinatorial testing with shielding parameters," in *Software Engineering Conference (APSEC)*, pp. 280–289, 2010.

[5] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI magazine*, vol. 17, no. 3, p. 37, 1996.

[6] T. J. Oyana, "A new-fangled fes-k-means clustering algorithm for disease discovery and visual analytics," *EURASIP Journal on Bioinformatics and Systems Biology*, vol. 2010, no. 1, p. 746021, 2010.

[7] M. Kantardzic, *Data mining: concepts, models, methods, and algorithms*. John Wiley & Sons, 2011.

[8] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender systems handbook*, vol. 1. Springer, 2011.

[9] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, pp. 89–100, ACM, 2007.

[10] S. Ruggieri, "Efficient c4. 5 [classification algorithm]," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 438–444, 2002.