Genetic Configuration Sampling

Learning a Sampling Strategy for Fault Detection of Configurable Systems

Jifeng Xuan
School of Computer Science
Wuhan University
Wuhan, China
jxuan@whu.edu.cn

Yongfeng Gu
School of Computer Science
Wuhan University
Wuhan, China
yongfenggu@whu.edu.cn

Zhilei Ren
School of Software
Dalian University of Technology
Dalian, China
zren@dlut.edu.cn

Xiangyang Jia
School of Computer Science
Wuhan University
Wuhan, China
jxy@whu.edu.cn

Qingna Fan
HY Cross-Domain
Wuhan, China
fanqn@crossdomain.cn

ABSTRACT

A highly-configurable system provides many configuration options to diversify application scenarios. The combination of these configuration options results in a large search space of configurations. This makes the detection of configuration-related faults extremely hard. Since it is infeasible to exhaust every configuration, several methods are proposed to sample a subset of all configurations to detect hidden faults. Configuration sampling can be viewed as a process of repeating a pre-defined sampling action to the whole search space, such as the *one-enabled* or *pair-wise* strategy.

In this paper, we propose genetic configuration sampling, a new method of learning a sampling strategy for configuration-related faults. Genetic configuration sampling encodes a sequence of sampling actions as a chromosome in the genetic algorithm. Given a set of known configuration-related faults, genetic configuration sampling evolves the sequence of sampling actions and applies the learnt sequence to new configuration data. A pilot study on three highly-configurable systems shows that genetic configuration sampling performs well among nine sampling strategies in comparison.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Search-based software engineering**; **Empirical software validation**;

KEYWORDS

Configuration sampling, fault detection, highly-configurable systems, genetic improvement, software configurations

ACM Reference Format:
Jifeng Xuan, Yongfeng Gu, Zhilei Ren, Xiangyang Jia, and Qingna Fan. 2018. Genetic Configuration Sampling: Learning a Sampling Strategy for Fault Detection of Configurable Systems. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208267

1 INTRODUCTION

Highly-configurable systems are widely deployed in our daily life. System designers provide many configuration options to meet the diverse requirements: an operating system (e.g., *Linux*) is required to adapt various types of hardware; a compiler (e.g., *Gcc*) is required to support multiple language features; a video decoder (e.g., *Mpeg*) is required to process different kinds of video streaming. The number of configuration options can reach several thousands. For instance, the C language compiler Gcc 7.3 contains 2,472 configuration options, which mainly focus on language features, compiling optimizations, and hardware setups [9]. In the command line of Gcc, each configuration option starts with `-`, such as `-ansi` for satisfying the C90 language standard or `-floop-nest-optimize` for the loop nest optimization. A user can quickly configure the compiler by choosing usable configuration options. In C programs, most of configuration options are implemented with the macro definition, such as `#ifdef-#else-#endif`.

Given a configurable system, a selection of configuration options is called a *configuration*, i.e., a combination of choices for configuration options. Many configuration-related faults are hidden behind the large number of feasible configurations. A combination of two seldom-checked configuration options may result in a system crash [29], or even a compiling error (considering the configuration options are encoded in the C macro definition) [18]. A straightforward way for fault detection is to exhaustively check all possible configurations. However, this leads to the problem of "combinatorial explosion". For a system with 100 binary configuration options, 1.27×10^{30} (i.e., 2^{100}) configurations exist. Meanwhile, checking many configurations is time-consuming. Even starting an operating system once takes several minutes.

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5764-7/18/07.
https://doi.org/10.1145/3205651.3208267

To address the problem of finding configuration-related faults, researchers have designed strategies to sample a subset of configurations rather than the whole set [12, 25]. Many sampling strategies for fault detection are studied, including *t-wise* [22], *one-enabled* [1], *random*. A sampling strategy can be viewed as a process of repeating a sampling action. For instance, the *pair-wise* (*t-wise* with $t = 2$) strategy aims to cover all combinations of pairs of configuration options: the process of *pair-wise* is a sequence of actions and each action is one combination of the values of two configuration options. Consider two binary configuration options a and b in a system, a sampling action of *pair-wise* can choose one configuration from $(\ldots,a,b,\ldots)$, $(\ldots,a,!b,\ldots)$, $(\ldots,!a,b,\ldots)$, and $(\ldots,!a,!b,\ldots)$. For a sampling strategy, the effectiveness of fault detection and the effort of sampling cannot be satisfied simultaneously. A comparative study by Medeiros et al. [18] shows that there exists a trade-off between the number of detected faults and the number of sampled configurations.

In this paper, we develop genetic optimization techniques to improve software configurations, which directly relate to the quality and the performance of highly-configurable systems. We propose genetic configuration sampling, a new method of evolving a sampling strategy for configurations. The evolved sampling strategy is not a sequence of uniform sampling actions like in *t-wise*, but a sequence of multiple sampling actions. For instance, an evolved strategy can interchangeably sample configurations via *t-wise* and *one-disabled*. Genetic configuration sampling encodes a sequence of sampling actions as a chromosome in the genetic algorithm. Given a set of known configuration-related faults, genetic configuration sampling learns a sequence of sampling actions and applies the learnt sequence to new configuration data. The idea of genetic configuration sampling is motivated by the algorithm family of *hyper-heuristics* [3]. We attempt to optimize the sequence of sampling actions rather than directly optimize the sampled configurations.

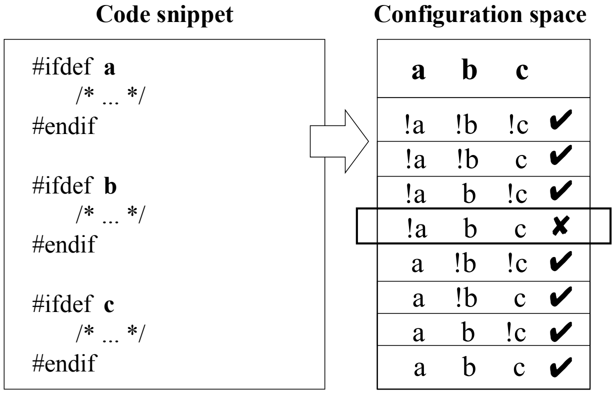
We conducted a pilot study on the configuration-related faults of three real-world systems, *Apache*, *BusyBox* and *Linux*. To evaluate the performance of genetic configuration sampling, we evolved a sampling strategy on the fault data of one system and apply the evolved strategy on another system. The result shows that genetic configuration sampling can perform well for fault detection of configurable systems, compared with existing sampling strategies.

This paper makes the following major contributions:

1. We proposed genetic configuration sampling, a new method of learning a sampling strategy for fault detection of highly-configurable systems. This method considers configuration sampling as a sequence of sampling actions and learns this sequence based on known configuration-related fault data. To the best of our knowledge, this is the first work that evolves a sampling strategy for fault detection of configurable systems.

2. We conducted a pilot study on three real-world configurable systems, *Apache*, *Busybox* and *Linux*. This study shows that genetic configuration sampling performs well among nine sampling strategies in comparison.

The remainder of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 presents the proposed method, genetic configuration sampling. Section 4 shows a pilot study on three configurable systems. Section 5 discusses the extension and technical details. Section 6 lists the related work and Section 7 concludes.



Code snippet

```
#ifdef a
    /* ... */
#endif

#ifdef b
    /* ... */
#endif

#ifdef c
    /* ... */
#endif
```

Configuration space

a	b	c	
!a	!b	!c	✔
!a	!b	c	✔
!a	b	!c	✔
!a	b	c	✘
a	!b	!c	✔
a	!b	c	✔
a	b	!c	✔
a	b	c	✔

Figure 1: Example of a configuration-related fault that is caused by a combinaion of three configuration options.

2 BACKGROUND AND MOTIVATION

We present the background of fault detection of configurable systems and hyper-heuristics, as well as the motivation of our work.

2.1 Fault Detection of Configurable Systems

In configurable systems, a crash or fault caused by interactions among configuration options is called a *configuration-related fault* [1, 18]. A specific combination of configuration options can expose configuration-related faults, such as violating compiling rules or triggering unexpected exceptions. Figure 1 lists an example of a configuration-related fault. Three candidate configuration options, a, b, and c, constitute a search space of eight combinations, i.e., eight configurations. In Figure 1, a and $!a$ indicate that a configuration option a is enabled and disabled, respectively. After checking all eight configurations, we find that only the configuration of disabling a and enabling b and c causes a fault.

To find a configuration-related fault, testers have to check each feasible configuration. A *brute force* strategy can guarantee the detection of all the configuration-related faults but its large search space of configurations cannot be covered in practice [10]. Even checking only one configuration, testers need to install and start the system. The process of checking a single configuration is time-consuming [24, 27]. To address the problem of configuration-related fault detection, various sampling strategies have been proposed. The insight of these strategies is to sample a subset of all configurations from the whole search space, i.e., detecting faults via representative configurations instead of all possible configurations.

We list four types of strategies of configuration sampling as follows.

(1) The *one-enabled* strategy [18] enables one configuration option and disables the remaining one in each configuration. In the case of Figure 1, this strategy can obtain three configurations, $(a,!b,!c)$, $(!a,b,!c)$, and $(!a,!b,c)$.

(2) The *one-disabled* strategy [1], which is the contrary to *One-enabled*, disables one configuration option and enables the remaining one in each configuration. This strategy can also provide three configurations, $(!a, b, c)$, $(a, !b, c)$, and $(a, b, !c)$.
(3) The *t-wise* strategy [22] enables or disables configuration options to cover all the combinations of each t options, where t is pre-defined ($t = 2, 3, \ldots$). Assuming $t = 2$, also called *pair-wise*, the sampled configurations are required to cover all the combinations of any pair of configuration options. *Pair-wise* requires at least four configurations; an example by *pair-wise* is $(!a, !b, !c)$, $(!a, b, c)$, $(a, !b, c)$, and $(a, b, !c)$.
(4) The *random* strategy is a straightforward way, which randomly enables or disables configuration options to obtain feasible configurations. The sampling result by *random* is unstable; an example is (a, b, c), $(!a, !b, c)$, and $(a, b, !c)$.

Note that in *t-wise*, a larger t requires a larger number of sampled configurations and can detect more faults. However, considering the scenario of a limited number of samples, a larger t may lose the opportunity of finding more faults.

2.2 Hyper-Heuristics

Hyper-heuristics are a family of heuristics, which leverages high-level heuristic framework to optimize low-level operators [3]. Different from directly optimizing solutions in meta-heuristics, hyper-heuristics only manipulate operators, such as two-swap or crossover; meanwhile, operators play the role of directly processing solutions. The mechanism of a high-level framework and its low-level operators in hyper-heuristics provides a hierarchical way to separate the optimization process with particular solutions. Following the development of meta-heuristics, many hyper-heuristics have emerged as solvers to complex optimization problems, such as hyper-heuristics based on hill-climbing by Özcan et al. [21], tabu search by Burke et al. [5], genetic programming by Nguyen et al. [20], and ant algorithms by Burke et al. [4].

2.3 Motivation

Existing sampling strategies are designed according to the experience of system designers or testers. Given a specific system, a configuration-related fault may not be detected by a particular strategy. For instance, the *one-disabled* strategy cannot cover the configuration $(!a, b, !c)$ in Figure 1. Meanwhile, the *t-wise* strategy may generate different subsets to satisfy the covering request; an off-the-shelf tool of *t-wise* follows pre-defined rules and cannot cover all possible configurations of *t-wise*, e.g., *pair-wise* in Section 2.1 cannot cover the configuration (a, b, c). On one hand, a small subset of sampling can reduce the cost of checking configurations but can lose the detected faults; on the other hand, a large subset can detect many faults but results in an enormous running cost.

In this paper, we view an existing sampling strategy as a sequence of repeating a single sampling action. Taking the *pair-wise* strategy as an example, its sampling action is to choose two configuration options at the same time and set their values. Then *pair-wise* repeats this action until all possible combinations of choosing two configuration options are covered. However, repeating a single action may lead to the lack of diversification. Several sampling actions can offset the weakness of each other.

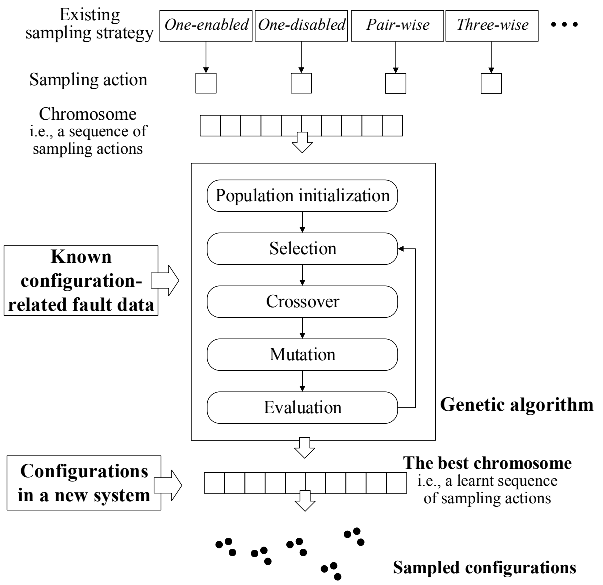

Existing sampling strategy
One-enabled
One-disabled
Pair-wise
Three-wise
• • •
Sampling action
Chromosome i.e., a sequence of sampling actions
Population initialization
Selection
Crossover
Mutation
Evaluation
Known configuration-related fault data
Genetic algorithm
Configurations in a new system
The best chromosome i.e., a learnt sequence of sampling actions
Sampled configurations

Figure 2: Overview of genetic configuration sampling.

In our work, the proposed method, genetic configuration sampling aims to learn a new sequence of multiple sampling actions, which come from existing sampling strategies, such as *one-enabled* and *pair-wise*. From the perspective of heuristics, genetic configuration sampling can be viewed as a hyper-heuristic, whose high-level framework is a genetic algorithm and low-level operators are sampling actions for configurations.

3 GENETIC CONFIGURATION SAMPLING

Genetic configuration sampling is a hyper-heuristic based on the genetic algorithm for detecting faults of configurable systems. The goal of genetic configuration sampling is to learn a sampling strategy by combining and evolving multiple existing sampling strategies. The sampling strategy is dynamically evolved in the framework of the genetic algorithm and updated according to the fitness value, i.e., the number of detected faults. We detail our method in this section.

3.1 Overview

Figure 2 shows the overview of our proposed method, genetic configuration sampling. The major difference between genetic configuration sampling and other sampling strategies is that genetic configuration sampling does not only consider one single sampling action; instead, several existing sampling actions are combined as a new sampling strategy, i.e., a sequence of sampling actions. Genetic configuration sampling adapts the framework of general genetic algorithms and encodes a sequence of sampling actions as a chromosome. A learnt sequence of sampling actions can be applied to the configuration options to generate a list of configurations. Due to the random factor, a given sequence of sampling actions may generate many different lists of configurations.

Table 1: Dataset of Three Real-World Systems, Apache, BusyBox and Linux

System	Domain	# Files	LoC	Configuration options	Faults
Apache	Web server application	362	144,768	700	12
BusyBox	Unix utility application	805	189,722	1,418	10
Linux	Operation system	37,520	12,594,584	26,427	37

To achieve an optimized (or near-optimized) sequence of sampling actions, genetic configuration sampling relies on a fitness function and a set of known configuration-related fault data. Given a particular chromosome, genetic configuration sampling repeats generating a list of sampled configurations to reduce the impact of randomization. The fitness function for evaluating a chromosome is to minimize the number of undetected faults by the sampled configurations. We define the fitness function as follows,

$$fitness(chromosome) = \frac{1}{m}\sum_{i=1}^{m} undetected_i$$

where m is the number of different trials of generated lists of configurations by one chromosome and $undetected_i$ is the number of undetected faults in the ith trial, i.e., the number of uncovered faults by the sampled configurations.

As shown in Figure 2, genetic configuration sampling learns a sampling strategy via the genetic algorithm on chromosomes. The best chromosome regarding the fitness value is selected as the learnt sequence of sampling actions. Then this sequence is applied to detect faults in potential configurations of a new system.

The genetic algorithm in genetic configuration sampling initializes a pool of chromosomes via randomization. Then the pool of chromosomes goes into the loop of four major steps in the genetic algorithm. First, in the selection step, top chromosomes with the best fitness values are selected as the current pool; second, in the crossover step, every two chosen chromosomes are mixed into two new chromosomes via a single-point crossover operator; third, in the mutation step, a part of chromosome is replaced by other sampling actions; fourth, in the evaluation step, all chromosomes are evaluated with a given fitness function. The four steps repeat until the pre-defined terminal condition; in genetic configuration sampling, the terminal condition is a given number of the maximum loops.

3.2 Sequence of Sampling Actions

In genetic configuration sampling, we employ seven sampling actions from existing and widely-used sampling strategies: *one-enabled*, *one-disabled*, *pair-wise*, *three-wise*, *four-wise*, *five-wise*, and *six-wise*. All these strategies have been empirically compared in a recent study by Medeiros et al. [18]. We briefly describe the seven sampling actions as follows. To facilitate the genetic algorithm, each sampling action in our work initializes from a random configuration. The *one-enabled* or *one-disabled* action randomly chooses one configuration option and enable or disable the configuration option. The *pair-wise*, *three-wise*, *four-wise*, *five-wise*, and *six-wise* actions can be unified as a *t-wise*, where t denotes 2, 3, 4, 5, and 6, respectively. A *t-wise* action randomly chooses t configuration options and sets the values of chosen configuration options to satisfy the *t-wise* sampling strategy.

Example. We use a single example to illustrate the process of genetic configuration sampling. Given a system with known configuration-related faults, genetic configuration sampling initializes a pool of chromosomes, each of which is a random sequence of sampling actions. After repeating four steps (selection, crossover, mutation, and evaluation) of the genetic algorithm, a final learnt sequence is obtained. Suppose the learnt sequence is *pair-wise*, *one-disabled*, *five-wise*, i.e., the length of chromosome is three. Then given a new system, genetic configuration sampling applies the learnt sequence to its configuration options. The process of applying the sequence can be as follows. First, the *pair-wise* action randomly selects two configuration options and selects one combination of their values (enabled or disabled) as the first configuration. Note that the selected configuration is required to be feasible, e.g., according to the dataset in this paper, two configuration options must come from the same source code file; meanwhile, the sampling action does not select duplicate configurations, e.g., the *pair-wise* action will try again if the values of two configuration options are already covered by a previous configuration. Second, the *one-disabled* action selects the second configuration to by disabling one configuration option. Third, the *five-wise* action selects the third configuration to cover a combination of values of five configuration options. From the fourth sampled configuration, the three sampling actions in the learnt sequence are applied repeatedly. The final list of sampled configurations is the sampling result for the new system.

4 A PILOT STUDY

We present a pilot study to preliminarily evaluate genetic configuration sampling.

4.1 Study Setup

We conducted this study on configuration-related faults of three real-world systems, *Apache*, *BusyBox* and *Linux*. Table 1 presents basic information of three systems. We followed the study by Medeiros et al. [18] to use their collected configuration options. The configuration options in these systems are extracted via automatic configuration tools and manual validation. As a pilot study, we have not considered all dependencies among configuration options. For instance, the constraints between two configuration options are omitted. Due to the code scale of these three systems, configuration options without constraints provide a complex application scenario for configuration sampling.

Table 2: Sequences of 10 Sample Actions that are Learnt from Three Systems

System	Sequence †
Apache	three-wise, one-enabled, five-wise, pair-wise, five-wise, four-wise, one-disabled, five-wise, pair-wise, six-wise
BusyBox	six-wise, four-wise, one-disabled, three-wise, four-wise, one-disabled, pair-wise, three-wise, one-enabled, one-disabled
Linux	pair-wise, six-wise, one-enabled, pair-wise, one-enabled, one-enabled, three-wise, three-wise, one-enabled, four-wise

† We denote a sampling action with its sampling strategy, e.g., *three-wise* is short for an action of the *three-wise* strategy.

As shown in Section 3.1, genetic configuration sampling learns a sequence of sampling actions based on a set of known configuration-related fault data. In this evaluation, we considered fault data of one system as known ones and learned the sequence via genetic configuration sampling; then we applied the learnt sequence to the configuration options of another system. We have not adapted general evaluation method, such as k-fold cross validation, because it is difficult to cut configurations into independent folds.

We implemented the method of genetic configuration sampling in Java JDK 1.8. All experiments were run on a PC with an Intel Core i7 3.60GHz CPU and 8 GByte memory. The package of implementation of genetic configuration sampling is online available.[1].

4.2 Results

The parameter setup of genetic configuration sampling is listed as follows. The length of chromosomes is 10; each chromosome is conducted based on seven sampling actions, i.e., actions from *one-enabled*, *one-disabled*, *t-wise* ($t = 2, \ldots, 6$). To evaluate the fitness of a particular chromosome, the chromosome is repeated 10 times to generate 100 sampled configurations for one system; the evaluation of a chromosome is the average of 50 trials. In the context of the genetic algorithm, the size of the chromosome pool is 30; the crossover rate is 90% and the mutation rate is 10%; the maximum generation of chromosomes is 20.

Table 2 presents the sequences of sampling actions, which genetic configuration sampling have learnt from the configuration-related data of three systems, *Apache*, *BusyBox*, and *Linux*. We have no clue which specific pattern the learnt sequences are following. According to our observation, it is difficult for human developers to design similar sequences like what genetic configuration sampling has learnt.

We compared genetic configuration sampling with eight existing sampling strategies, including the random strategy and seven sampling strategies that contain the sampling actions in genetic configuration sampling. All these sampling strategies can be found in [18]. We individually ran each sampling strategy 50 times and calculated the average.

Tables 3, 4, and 5 present the average numbers of detected faults by genetic configuration sampling and eight sampling strategies under comparison. We use *Apache*⇒*BusyBox* to denote learning a sequence from the *Apache* data and applying to the ⇒*BusyBox* data. A table cell with − indicates that all potential configurations of the sampling strategy have already been checked.

As shown in Tables 3, 4, and 5, genetic configuration sampling performs well when sampling 100 configurations for all three systems under evaluation. In *Apache*, genetic configuration sampling that learns from the *BusyBox* data reaches 11.80, the highest number of detected faults; in *BusyBox*, genetic configuration sampling that learns from the *Apache* data reaches 8.06, the second highest number of detected faults; in *Linux*, genetic configuration sampling that learns from the *BusyBox* data reaches 20.14, the second highest number of detected faults. In *BusyBox* and *Linux*, the *pair-wise* and *one-disabled* strategies obtain the highest numbers of detected faults, respectively. We found that results of these two strategies are unstable. For instance, *one-disabled* for the *BusyBox* data detects only 5.60 faults, which is the lowest result among all strategies under evaluation. The result of genetic configuration sampling seems stable.

Genetic configuration sampling have not obtained the best result when sampling a small number of configurations. For instance, sampling 10 configurations by genetic configuration sampling in *Apache* is lower than seven but one existing strategies. However, sampling a small number (like 10) of configurations highly relies on the seeding configuration, which indicates the starting point of the strategy. We consider that finding faults is more important. Meanwhile, existing strategies for sampling a small number of configurations is also unstable.

From the results by genetic configuration sampling, we can find that known data from different systems lead to variable results. In both *Apache* and *Linux*, learning from the *BusyBox* achieves better results than from the other systems; in both *Apache* and *BusyBox*, learning from the *Linux* achieves worse results than from the other systems. The difference among systems will be discussed in Section 5.

To sum up, genetic configuration sampling behaves well in the evaluation of detected faults on three systems. The learnt strategy in genetic configuration sampling relies on a set of known fault data. The result in this pilot study shows that the known data of one system can be employed to guide the configuration sampling of another system.

5 DISCUSSION

In this paper, we propose a new method of evolving a sampling strategy for fault detection of configurable systems. The presented study is preliminary. We discuss our study as follows.

System data transfer. To evaluate genetic configuration sampling, we learned a sequence of sampling actions from one system and applied the learnt sequence to another system. The assumption behind is that different systems share the same data distribution; however, this assumption is hardly satisfied in practice. The performance of a learnt sequence from one system can be hurt if the sequence is applied to another system. The method of recovering

[1] Genetic configuration sampling, http://cstar.whu.edu.cn/p/gcs/.

Table 3: Average Number of Detected Faults on the Apache System

Sampling strategy	Size of sampled configurations									
	10	20	30	40	50	60	70	80	90	100
BusyBox⇒Apache	3.88	6.64	8.68	10.34	11.08	11.34	11.46	11.58	11.72	**11.80**
Linux⇒Apache	4.28	7.20	9.32	10.74	11.02	11.20	11.26	11.32	11.46	**11.54**
One-enabled	1.32	2.44	3.52	4.24	5.28	6.44	7.52	8.22	9.00	9.70
One-disabled	6.48	7.84	8.64	9.16	9.60	9.84	10.18	10.44	10.62	10.80
Pair-wise	4.88	7.46	9.48	10.30	11.04	11.62	-	-	-	-
Three-wise	5.12	7.32	8.66	9.36	9.84	10.02	10.38	10.52	10.64	10.76
Four-wise	4.76	6.68	7.78	8.48	9.02	9.44	9.74	9.96	10.24	10.46
Five-wise	5.14	6.70	7.56	8.02	8.52	8.76	8.90	9.12	9.26	9.42
Six-wise	4.96	6.34	6.92	7.30	7.56	7.84	8.00	8.18	8.30	8.48
Random	4.98	7.22	8.42	9.12	9.58	9.92	10.18	10.40	10.50	10.68

Table 4: Average Number of Detected Faults on the BusyBox System

Sampling strategy	Size of sampled configurations									
	10	20	30	40	50	60	70	80	90	100
Apache⇒BusyBox	1.82	3.76	5.26	5.82	6.28	6.72	7.14	7.36	7.68	**8.06**
Linux⇒BusyBox	1.66	2.88	4.40	5.42	6.06	6.52	6.96	7.48	7.68	**7.86**
One-enabled	1.34	2.10	2.68	3.38	3.92	4.50	5.12	5.54	5.96	6.52
One-disabled	0.72	1.54	2.26	2.86	3.18	3.70	4.14	4.52	5.04	5.60
Pair-wise	2.78	4.40	5.90	7.22	8.14	-	-	-	-	-
Three-wise	1.86	3.18	3.84	4.26	4.72	5.02	5.30	5.76	5.98	6.16
Four-wise	2.02	3.20	3.96	4.44	4.92	5.36	5.72	6.12	6.34	6.50
Five-wise	2.38	3.62	4.42	4.78	5.26	5.70	6.02	6.28	6.52	6.68
Six-wise	2.26	3.62	4.18	4.62	4.92	5.24	5.46	5.62	5.82	5.98
Random	2.18	3.42	4.18	4.84	5.20	5.78	6.16	6.44	6.84	7.08

Table 5: Average Number of Detected Faults on the Linux System

Sampling strategy	Size of sampled configurations									
	10	20	30	40	50	60	70	80	90	100
Apache⇒Linux	2.66	4.88	6.98	9.02	10.66	12.42	14.34	16.22	17.86	**19.48**
Busybox⇒Linux	3.00	5.32	7.36	9.76	11.30	13.32	15.22	16.68	18.50	**20.14**
One-enabled	2.66	4.22	5.56	6.68	7.72	8.74	9.78	10.70	11.74	12.54
One-disabled	5.18	8.34	11.00	13.20	14.90	16.42	18.00	19.42	20.72	22.14
Pair-wise	3.52	6.08	8.08	9.98	12.04	13.68	14.98	16.68	17.84	19.14
Three-wise	3.56	6.44	8.48	10.42	12.44	13.98	15.30	16.42	17.74	18.90
Four-wise	3.70	6.32	8.38	10.12	11.40	12.86	13.92	14.98	15.88	16.96
Five-wise	3.52	5.38	7.20	8.58	9.86	10.88	11.84	12.74	13.74	14.50
Six-wise	3.64	5.92	7.72	8.84	9.80	10.76	11.48	12.24	12.98	13.36
Random	3.68	6.22	8.20	9.72	11.38	12.98	13.98	14.94	16.24	17.28

the data difference between two systems is called *data transfer*. We will explore the influence of data transfer in future work.

Random factor. In genetic configuration sampling, as well as general genetic algorithms, randomization plays an important role in searching for new chromosomes. Different from existing sampling strategies, such as *t-wise*, genetic configuration sampling leverages the random factor to connect pre-defined sampling actions. This leads to the high performance and also results in unstable results. Existing meta-heuristics like instance reduction [6, 11] may help stabilize the disturbance by the random factor; a larger study of comparison may help understand the random factor in genetic configuration sampling.

Running time. The process of learning a sampling strategy of genetic configuration sampling takes two to five minutes. This leads to larger running time than existing sampling strategies. However, as mentioned in Section 2, the major cost of configuration sampling is the time cost of checking one configuration and then exhausting all configurations. The running time of learning a sequence of sampling actions is not expensive.

What makes genetic configuration sampling perform well. In genetic configuration sampling, we optimize the sequence of sampling actions rather than directly optimize the sampled configurations. This design separates the optimization framework with particular sampling actions. That is, a chromosome directly relates to sampling not to configurations. Beyond genetic configuration sampling, it is possible to further optimize a sampling action; this may find a better learnt sequence than only optimizing the sequence of sampling actions.

Involving constraints for configuration options. Our study has not evaluated sampling strategies on all datasets. In practice, configuration options have many complex dependencies, e.g., constraints, global dependencies, and building scenarios [18]. These complex features of configuration options can result in more potential research questions for sampling strategies.

6 RELATED WORK

Sampling strategies are common methods to solve the *faults detection* problem in configurable systems. The *t-wise* sampling strategy, which derives from the domain of combinatorial testing [15], assumes that most of faults are caused by specific combinations of configuration options. This assumption has been confirmed by many empirical studies [13, 14, 28]. Cohen et al. [7] first implemented the AETG tool to solve the *t-wise* test cases generation problem. AETG greedily generates combinations to cover the most uncovered *t* options each time, until the selected combinations can satisfy the *t-wise* request. Borazjany et al. [2] proposed the ACTS tool by modeling a suitable input space. ACTS extends existing tools by adding several greedy algorithms, such as IPOG [16] and IPOGD [17]. Recently, Garciarena and Santana [8] have optimized the selection of compiler flags via learning and exploiting flag interactions. Beside the *t-wise* strategies, *one-enabled* by Medeiros et al. [18] and *one-disabled* by Abal et al. [1] are two intuitive sampling strategies. Both strategies focus on the system status that only one configuration option is enabled or disabled. Existing studies like [18] show that *one-enabled* and *one-disabled* can trigger many configuration-related faults.

Apart from detecting faults, the problem of performance prediction is widely-studied for configurable systems. *Performance prediction* aims to infer the system performance (i.e., a non-functional property, such as response time, memory consumption, and throughput) based on selected configurations [24, 27]. Guo et al. [10] proposed a learnable model based on a decision tree algorithm, CART. Their model predicts the performance of unmeasured configurations by revealing the correlation between known configurations and their performance. Their experiments on six real-world projects demonstrated that the CART model reaches an average of 94% accuracy. Their following work [26] shows that incorporating the Bagging method can achieve better results of learnable models, such as CART, Random Forest, and SVM.

Sarkar et al. [23] compared the efficiency between two typical sampling strategies, *progressive* and *projective*, on performance prediction. They improved the projective sampling method with a novel feature-frequency heuristic method, which outperforms the traditional *t-wise* method. Recently, Nair et al. [19] proposed *rank-based* performance prediction, which transforms the numerical regression problem into a practical problem, i.e., ranking. Valov et al. [27] trained a transfer model of performance prediction on cross machine platforms. They built a prediction model on one platform and then used linear transformation techniques to predict the performance on other platforms. Experiments on three systems across 23 platforms showed that the transfer model can achieve less than 10% mean relative error on cross-platform performance prediction.

7 CONCLUSIONS

We propose genetic configuration sampling, a new method of learning a sampling strategy for fault detection of highly-configurable systems. Genetic configuration sampling aims to evolve a sequence of sampling actions based on known configuration-related faults of one system and apply to configurations of another system. This evolved sequence can leverage the sampling actions of several existing sampling strategies, such as *one-enabled* and *pair-wise*. In genetic configuration sampling, a genetic algorithm is used to direct these sampling actions, rather than search or optimize a list of specific sampled configurations. We employ genetic improvement techniques to optimize software configurations, which can improve the quality and the performance of highly-configurable systems.

In future work, we plan to design a new method to make genetic configuration sampling apply its learnt sequence to the same system; that is, we expect that learning and applying genetic configuration sampling share different configurations of the same system. We also would like to conduct a large study to explore the performance of genetic configuration sampling in different systems.

ACKNOWLEDGMENTS

The authors would like to thank Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel for sharing the data of configuration-related faults.

The work is partly supported by the National Natural Science Foundation of China under Grant No.: 61502345 and 61772107, the Young Elite Scientists Sponsorship Program By CAST under Grant

No.: 2015QNRC001, and the Technological Innovation Projects of Hubei Province under Grant No.: 2017AAA125.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the Linux kernel: A qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 421–432. https://doi.org/10.1145/2642937.2642990

[2] Mehra N. Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and Rick Kuhn. 2012. Combinatorial Testing of ACTS: A Case Study. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. 591–600. https://doi.org/10.1109/ICST.2012.146

[3] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724.

[4] Edmund K. Burke, Graham Kendall, Dario Landa Silva, Ross O'Brien, and Eric Soubeiga. 2005. An ant algorithm hyperheuristic for the project presentation scheduling problem. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2005, 2-4 September 2005, Edinburgh, UK*. 2263–2270. https://doi.org/10.1109/CEC.2005.1554976

[5] Edmund K. Burke, Sanja Petrovic, and Rong Qu. 2006. Case-based heuristic selection for timetabling problems. *J. Scheduling* 9, 2 (2006), 115–132. https://doi.org/10.1007/s10951-006-6775-y

[6] Zongzheng Chi, Jifeng Xuan, Zhilei Ren, Xiaoyuan Xie, and He Guo. 2017. Multi-Level Random Walk for Software Test Suite Reduction. *IEEE Comp. Int. Mag.* 12, 2 (2017), 24–33. https://doi.org/10.1109/MCI.2017.2670460

[7] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatiorial Design. *IEEE Trans. Software Eng.* 23, 7 (1997), 437–444. https://doi.org/10.1109/32.605761

[8] Unai Garciarena and Roberto Santana. 2016. Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions. In *Genetic and Evolutionary Computation Conference, GECCO 2016, Denver, CO, USA, July 20-24, 2016, Companion Material Proceedings*. 1159–1166. https://doi.org/10.1145/2908961.2931696

[9] GNU. 2018. Using the GNU Compiler Collection (GCC). Retrieved March 28, 2018 from https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

[10] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 301–311. https://doi.org/10.1109/ASE.2013.6693089

[11] He Jiang, Jifeng Xuan, and Zhilei Ren. 2010. Approximate backbone based multilevel algorithm for next release problem. In *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*. 1333–1340. https://doi.org/10.1145/1830483.1830730

[12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. 46–55. https://doi.org/10.1145/2362536.2362547

[13] D. Richard Kuhn and Michael J. Reilly. 2002. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*. 91–95. https://doi.org/10.1109/SEW.2002.1199454

[14] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Software Eng.* 30, 6 (2004), 418–421. https://doi.org/10.1109/TSE.2004.24

[15] Rick Kuhn, Yu Lei, and Raghu Kacker. 2008. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional* 10, 3 (2008), 19–23. https://doi.org/10.1109/MITP.2008.54

[16] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA*. 549–556. https://doi.org/10.1109/ECBS.2007.47

[17] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Softw. Test., Verif. Reliab.* 18, 3 (2008), 125–148. https://doi.org/10.1002/stvr.381

[18] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 643–654. http://doi.acm.org/10.1145/2884781.2884793

[19] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 257–267. https://doi.org/10.1145/3106237.3106238

[20] Su Nguyen, Mengjie Zhang, and Mark Johnston. 2011. A genetic programming based hyper-heuristic approach for combinatorial optimisation. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Dublin, Ireland, July 12-16, 2011*. 1299–1306. https://doi.org/10.1145/2001576.2001752

[21] Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. 2006. Hill Climbers and Mutational Heuristics in Hyperheuristics. In *9th International Conference on Parallel Problem Solving from Nature, PPSN IX, Reykjavik, Iceland, September 9-13, 2006, Procedings*. 202–211. https://doi.org/10.1007/11844297_21

[22] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. 459–468. https://doi.org/10.1109/ICST.2010.43

[23] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 342–352. https://doi.org/10.1109/ASE.2015.45

[24] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 167–177. https://doi.org/10.1109/ICSE.2012.6227196

[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90, 000 #ifdefs Issue. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 421–432. https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler

[26] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2015. Empirical comparison of regression methods for variability-aware performance prediction. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. 186–190. https://doi.org/10.1145/2791060.2791069

[27] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 39–50. https://doi.org/10.1145/3030207.3030216

[28] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. 2016. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. *Information & Software Technology* 76 (2016), 65–80. https://doi.org/10.1016/j.infsof.2016.04.016

[29] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 910–913. https://doi.org/10.1145/2786805.2803206