The Journal of Systems & Software 190 (2022) 111353

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

CIT-daily: A combinatorial interaction testing-based daily build process☆

Hanefi Mercan, Atakan Aytar, Giray Coskun, Dilara Mustecep, Gülsüm Uzer, Cemal Yilmaz *

Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey

ARTICLE INFO

Article history:
Received 15 September 2021
Received in revised form 22 April 2022
Accepted 26 April 2022
Available online 12 May 2022

Keywords:
Daily build processes
Combinatorial interaction testing
Covering arrays
Software testing

ABSTRACT

In this work, we introduce an approach, called *CIT-daily*, which integrates combinatorial interaction testing (CIT) with the daily build processes to systematically test the interactions between the factors/parameters affecting the system's behaviors, on a daily basis. We also develop a number of CIT-daily strategies and empirically evaluate them on highly-configurable systems. The first strategy tests the same t-way covering array every day throughout the process, achieving a t-way coverage on a daily basis by covering each possible combination of option settings for every combination of t options. The other strategies, on the other hand, while guaranteeing a t-way coverage on a daily basis, aim to cover higher order interactions between the configuration options over time by varying the t-way covering arrays tested. In the experiments, we observed that the proposed approach significantly improved the effectiveness (i.e., fault revealing abilities) of the daily build processes; randomizing the coverage of higher order interactions between the configuration options while guaranteeing a base t-way coverage every day, further improved the effectiveness; and the more the higher order interactions covered during the process, the higher the fault revealing abilities tended to be.

1. Introduction

A daily build process is a process where the latest version of a software system under development (SUD) is downloaded, configured, built, and tested on a daily basis (typically during off-work hours). The ultimate goal of this process is to reveal the defects regarding the fundamental functionalities of the system as early as possible, so that the turn-around time for fixes is reduced as much as possible (Fowler and Foemmel, 2006). Consequently, the daily build processes have been extensively used for software testing (Memon et al., 2003; Fowler and Foemmel, 2006).

We, however, observe that systematic testing of the interactions between the factors/parameters that affect the system's behaviors, is often overlooked for in the daily build processes. For example, for configurable software systems, the daily build processes are typically carried out by running the test cases on the default configuration of the system every day. Therefore, the opportunity of revealing in a timely manner the failures that are caused by the interactions of configuration options, is generally lost.

In this work, we conjecture that the effectiveness of the daily build processes can significantly be improved, if the interactions between the factors/parameters affecting the system's behaviors are systematically tested throughout the process. Note that the exhaustive testing of all possible interactions is typically not feasible as the number of interactions to be tested grows exponentially with the number of factors/parameters.

We, in particular, present an approach, called *CIT-daily*, which integrates combinatorial interaction testing (CIT) with the daily build processes. CIT-daily systematically samples the interactions and tests only the selected interactions on a daily basis. To evaluate the proposed approach, we solely focus on the configurable software systems in the paper. However, CIT-daily can readily be applicable to any type of system as long as the system has multiple interacting factors/parameters, i.e., in all the scenarios where CIT is applicable, including the systematic testing of the input parameters (Jarman and Smith, 2019; Rao and Li, 2021), software product lines (Qian et al., 2018; Lopez-Herrejon et al., 2015), network protocols (Choi, 2017), and graphical user interfaces (Michaels et al., 2020; Mirzaei and Garcia, 2016).

At a very high level, the CIT-daily process is carried out on a daily basis as follows: (1) the latest version of the system under development is obtained from its code repository; (2) the configuration space of the system is systematically sampled, producing

☆ Editor: A. Bertolino.

* Corresponding author.
E-mail addresses: hanefimercan@sabanciuniv.edu (H. Mercan), atakanaytar@sabanciuniv.edu (A. Aytar), giraycoskun@sabanciuniv.edu (G. Coskun), dilaramustecep@sabanciuniv.edu (D. Mustecep), guzer@sabanciuniv.edu (G. Uzer), cyilmaz@sabanciuniv.edu (C. Yilmaz).

https://doi.org/10.1016/j.jss.2022.111353
0164-1212/

a set of configurations to be tested; and (3) for each configuration selected, the system is configured, built, and tested by running its test suite on the configuration.

To systematically sample the configuration spaces, CIT-daily uses one of the most frequently used CIT objects, namely *covering arrays* (Yilmaz et al., 2013b). Given a configuration space for the SUD and a coverage strength t, a t-way covering array is a set of configurations carefully selected from the configuration space to achieve a full coverage under the *t-way coverage criterion* where each possible combination of option settings for every combination of t configuration options appears at least once (Yilmaz, 2012; Fouché and Porter, 2009; Lei et al., 2008a).

The basic justification for using t-way covering arrays is that they (under certain assumptions) can efficiently and effectively reveal all system failures caused by the interactions of t or fewer parameters (Yilmaz et al., 2013b). Furthermore, many of the empirical studies suggest that a majority of the option-related failures are caused by the interactions between only a small number of options. That is, t, compared to the number of configuration options, is generally quite small in practice (typically, $t = 2$ or 3 (Cohen et al., 1997; Alazzawi and Rais, 2019; Mohammad and Valepe, 2019; Charbachi et al., 2017)). And, when t is fixed, as the number of configuration options increases, the size of a covering array represents an increasingly smaller portion of the entire configuration space. Consequently, covering arrays have been extensively used for software testing in a wide spectrum of application domains (Jarman and Smith, 2019; Rao and Li, 2021; Yilmaz et al., 2006; Qian et al., 2018; Michaels et al., 2020; Qi et al., 2018; Choi, 2017).

In this work, we integrate covering arrays with the daily build processes. To this end, we have, indeed, developed a number of different strategies (Section 3), namely *fixed coverage*, *randomized coverage*, *opportunistic coverage*, *guaranteed coverage*, and *optimized coverage*. All of these strategies deliberately use the existing covering array constructors to improve their applicabilities in practice, such that not only the standard covering arrays, but also the other types of CIT objects, such as cost-aware covering arrays (Demiroz and Yilmaz, 2012), test case-aware covering arrays (Yilmaz, 2012), and U-CIT objects (Mercan et al., 2022), can be integrated with the daily build processes as long as an appropriate constructor is provided.

More specifically, the fixed coverage strategy simply uses the same t-way covering array for testing on a daily basis (Section 3.1), whereas the randomized coverage strategy attempts to use a different t-way covering array every day, guaranteeing the coverage of the t-way combinations while randomizing the coverage of the higher strength combinations (Section 3.2). The remaining strategies, on the other hand, take as input two coverage strengths, namely t_1 and t_2 where $t_1 < t_2$, and aim to obtain a full t_2-way coverage over time during the CIT-daily process while guaranteeing a t_1-way coverage on a daily basis. To this end, the opportunistic coverage strategy aims to opportunistically maximize the coverage of the t_2-way combinations by greedily testing a t_1-way covering array every day, which, compared to the other candidates considered, covers the most number of previously uncovered t_2-way combinations. The guaranteed (Section 3.4) and the optimized (Section 3.5) coverage strategies, however, ensure to obtain a full t_2-way coverage either within a given number of days (as is the case in the guaranteed coverage strategy) or within a given daily overhead limit (as is the case in the optimized coverage strategy).

To evaluate the proposed approach, we carried out a series of experiments by using a number of highly configurable, real-life software systems as our subject applications (Section 4). We observed that (1) the proposed approach significantly improved the effectiveness (i.e., fault revealing abilities) of the daily build processes, compared to the current daily build practices where all the test cases are executed on the default configurations of the systems; (2) randomizing the coverage of higher strength combinations (i.e., t_2-way combinations) while guaranteeing a base t_1-way coverage on a daily basis, further improved the effectiveness; and (3) the more the higher strength combinations were covered during the process, the higher the fault revealing abilities tended to be.

The contributions of this paper can be summarized as follows:

- the introduction of a CIT-based daily build process, which allows practical constraints, such as duration limits and the daily overhead limits, into account while systematically covering the interactions between various factors/parameters affecting the system's behaviors on a daily basis by integrating covering arrays with the daily build processes,
- a number of different CIT-based testing strategies, which deliberately leverage the existing CIT constructors to improve the applicability of the proposed strategies in practice,
- empirical evaluations of the proposed strategies on evolving, real-life software systems, demonstrating the fault revealing abilities of these strategies.

The remainder of the paper is organized as follows: Section 2 provides background information covering arrays; Section 3 introduces the proposed approach as well as the different strategies we developed; Section 4 presents the empirical studies; Section 5 discusses threats to validity; Section 6 discusses related work; and Section 7 concludes with some future work ideas.

2. Covering arrays

A t-way covering array (CA) constructor takes as input a *configuration space model* and a *coverage strength* t. In its simplest form, the model includes a set of configuration options and their settings together with system-wide inter-option constraints (if any), which invalidate certain combinations of option settings as not all combinations may valid in practice. In effect, the configuration space model implicitly defines a space of valid configurations for testing. The coverage strength t, on the other hand, specifies the coverage criterion to be satisfied by sampling the configuration space.

Definition 1. Given a configuration space model, a t-tuple is an ordered set of option-setting pairs for a combination of t distinct options, which does not violate any of the system-wide constraints stated in the model.

Definition 2. Given a configuration space model with n options, a configuration is an n-tuple.

Definition 3. Given a configuration space model and a coverage strength t, a set of configurations is called a t-way covering array if it satisfies the t-way coverage criterion, i.e., if each t-tuple appears in at least one of the configurations included in the set.

Definition 4. The size of a covering array is the number of configurations included in the array.

Table 1 represents an example 2-way covering array created for a hypothetical system that supports two operating systems (OS), namely Windows and MacOS; three web browsers, namely Chrome, Firefox, and Safari; and two database management systems, namely MySQL and Oracle. Consequently, the configuration space model of the system contains three options; one with three settings (i.e., browser = {Chrome, Firefox, Safari}) and the remainings with two settings each (i.e., OS = {Windows, MacOS}

Table 1
An example 2-way covering array.

OS	Browser	Database
Windows	Chrome	Oracle
MacOS	Chrome	MySQL
Windows	Firefox	MySQL
MacOS	Firefox	Oracle
MacOS	Safari	MySQL
MacOS	Safari	Oracle

System-wide constraint:
OS = Windows $\implies$ browser $\neq$ Safari

and database = {MySQL, Oracle}). Furthermore, the model also includes "OS = Windows $\implies$ browser $\neq$ Safari" as a system-wide constraint, indicating that OS = Windows and browser = Safari is an invalid combination as Safari is not supported on Windows. Since $t = 2$ in the covering array given in Table 1, for any given pair of options, the respective columns include each valid combination of settings for the selected options at least once. Note further that none of the selected configurations violates the system-wide constraint and that the size of the covering array is 6 (i.e., it contains 6 configurations). To reduce the cost of testing (under certain cost models), covering arrays are, indeed, computed, such that their sizes are minimized as much as possible.

Determining the coverage strength (t) to be used with the covering arrays is still an open question. Many empirical studies in the literature, on the other hand, demonstrate that a majority of the option-related failures are generally revealed when $t \leq 6$, with $t = 2$ being the most commonly used strength in practice (Hartman, 2005; Kuhn et al., 2010).

One well-known and frequently-used mechanism with covering arrays is the *seeding mechanism* (Gladisch et al., 2020). In this context, a *seed* is a set of partially or fully specified configurations. Given a seed, a t-way covering array is constructed around the seed. That is, all the t-tuples present in the seed are considered to have already been covered and additional configurations are added only to cover the missing t-tuples. The seeding mechanism has been used for various purposes in the literature, including to guarantee the inclusion of certain configurations in the covering arrays (Ma et al., 2018) and to compute higher strength covering arrays from lower strength ones (Galinier et al., 2017). In this work, we, on the other hand, use the seeding mechanism in a novel way to guarantee the coverage of the higher strength tuples in a sequence of lower strength covering arrays (not just in one higher strength covering array) during the CIT-daily process by using the non-overlapping subsets of higher strength covering arrays as seeds to compute lower strength covering arrays (Sections 3.4–3.5).

In the remainder of the paper, we use the computational primitive *computeCA*(*model*, t, *seed*) to compute a t-way covering array around the seed *seed* (which is an optional parameter) for the given configuration space model *model*. Furthermore, this primitive is assumed to generate randomized covering arrays in the sense that, given the same arguments, it would potentially produce a different t-way covering array. Note that almost all of the existing covering array constructors, such as *Jenny* (Jenkins, 2005) and *ACTS* (Yu et al., 2013), implement this primitive.

3. CIT-daily

CIT-daily takes as input a software system under development (SUD), a configuration space model specifying a valid configuration space for testing, and a coverage criterion to be satisfied throughout the CIT-daily process. The goal is to obtain a full coverage under the given coverage criterion by systematically testing the interactions between the configuration options on a daily basis, so that option-related failures can be discovered as early as possible, which, in turn, can reduce the turnaround time for fixes.

Algorithm 1 The CIT-daily process

```
1: procedure CIT-DAILY(sud, model, strategy)
2:     for each day at a predetermined time do
3:         sud.checkout()
4:         ca ← strategy.run(model)
5:         for each c ∈ ca do
6:             sud.configure(c)
7:             sud.build()
8:             sud.test()
```

The coverage criterion in CIT-daily can specify not only the coverage properties to be satisfied on a daily basis, but also the ones to be satisfied over a period of days. To make the CIT-daily framework extensible, the coverage criteria are implemented in the form of strategies that can easily be plugged in and out of the framework.

At a very high level, CIT-daily operates as follows (Algorithm 1): every day (1) the latest version of the SUD is checked out from its code repository (line 3); (2) the configuration space is systematically sampled to obtain full coverage under the given coverage criterion either by computing a covering array or by using a pre-computed covering array (line 4); and (3) for each and every configuration included in the covering array, the SUD is configured, built, and tested by running its test suite on the configuration (lines 5–8).

We have developed a number of different CIT strategies to be used with CIT-daily. Next, we discuss these strategies from the simplest one to the most sophisticated one. Each strategy was introduced to further improve the effectiveness of the CIT-daily process by enhancing the coverage properties throughout the process while guaranteeing a base coverage on a daily basis.

3.1. Strategy 1: Fixed coverage

Our first strategy (Algorithm 2) takes as input a coverage strength t, computes a t-way covering array (line 3), and uses the same array every day to test the SUD. Note that this basic strategy, which is inspired from the standard CIT practices (Kuhn et al., 2010; Jarman and Smith, 2019; Mukelabai and Nešić, 2018; Li et al., 2019), aims to reveal all failures caused by the interactions of t or fewer parameters on a daily basis.

Algorithm 2 Fixed Coverage Strategy

```
   global ca ← null
1: procedure FIXEDCOVSTRATEGY(model, t)
2:     if ca is null then
3:         ca ← computeCA(model, t)
4:     return ca
```

3.2. Strategy 2: Randomized coverage

The randomized coverage strategy (Algorithm 3), as was the case with the fixed coverage strategy, takes as input a coverage strength t. Unlike the former strategy, however, this strategy uses a different t-way covering array every day for testing (line 2). By doing so, the randomized coverage strategy aims to reveal more defects by varying the configurations tested throughout the CIT-daily process while guaranteeing a t-way coverage on a daily basis.

Note that although a different t-way covering array is tested every day, each of these covering arrays is guaranteed to cover all of the valid t-tuples. However, as different set of configurations are selected to obtain the t-way coverage, the coverage of the higher strength tuples (i.e., the t'-tuples where $t' > t$) may vary from one day to another. And, this, in turn, can help reveal more failures over time by testing a larger number of distinct higher order interactions between configuration options.

Algorithm 3 Randomized Coverage Strategy

```
1: procedure RandomizedCovStrategy(model, t)
2:     ca ← computeCA(model, t)
3:     return ca
```

3.3. Strategy 3: Opportunistic coverage

The opportunistic coverage strategy (Algorithm 4) takes as input two coverage strengths, namely t_1 and t_2, such that $t_1 < t_2$, and opportunistically aims to obtain a full t_2-way coverage over time while guaranteeing a t_1-way coverage on a daily basis.

To this end, the opportunistic coverage strategy computes n different t_1-way covering arrays on a given day (lines 3–5) and then, among all of these newly computed covering arrays, selects the one that covers the most number of previously uncovered t_2-tuples (line 6) for testing. Note that the latter step is carried out in an attempt to reduce the number of days required to achieve a full t_2-way coverage. When a t_2-way coverage is achieved, the entire process is repeated from scratch (lines 7–8), potentially producing a different sequence of covering arrays for testing.

3.4. Strategy 4: Guaranteed coverage

The previous strategy (Section 3.3) opportunistically attempts to achieve a higher strength coverage over time. The guaranteed coverage strategy, on the other hand, takes as input a positive integer k and two coverage strengths, namely t_1 and t_2 ($t_1 < t_2$), and computes a plan, which guarantees to obtain a t_2-way coverage in exactly k days while ensuring a t_1-way coverage on a daily basis. In the remainder of the paper, a static collection of covering arrays is referred to as a *plan*. And, a *schedule* refers to a plan where each covering array in the plan is scheduled to be executed on a distinct day.

To this end, the guaranteed coverage strategy (Algorithm 5) first computes a t_2-way covering array (line 3), then divides this array into k equal- or almost equal-sized, non-overlapping partitions (line 4), and finally uses each partition as a seed to compute a t_1-way covering array (line 6). Note that the function *divide*(*ca*, *size*) (line 4) divides a given covering array *ca* into a minimum number of non-overlapping partitions, such that all the partitions are of size *size*, except possibly for one, the size of which can be slightly lower than *size* due to the divisibility issues. The output is a k-day plan, in which each t_1-way covering array computed is scheduled to be tested on a different day (lines 8–9).

Note that the computed plan guarantees to obtain t_2-way coverage in k days as the collection of all the configurations tested throughout the plan is guaranteed to include all the configurations in the initially created t_2-way covering array, the parts of which are used to form the seeds. Furthermore, once a full t_2-way coverage is achieved, the entire process is repeated from scratch (lines 2–7), potentially producing a different plan for testing.

Algorithm 4 Opportunistic Coverage Strategy

```
    ▷ History of the covering arrays tested so far
    global history ← {}
1:  procedure OpporCovStrategy(model, t1, t2, n)      ▷ Collection of candidate covering arrays to be used
2:      pool ← {}
3:      for n times do
4:          ca ← computeCA(model, t1)
5:          pool ← pool ∪ ca
        ▷ Given history, pick the ca ∈ pool that covers the most number of previously uncovered t2-tuples
6:      ca ← bestCA(history, model, pool, t2)
7:      if history obtains t2-way coverage then
8:          history ← {}
9:      else
10:         history ← history ∪ ca
11:     return ca
```

Algorithm 5 Guaranteed Coverage Strategy

```
    global plan ← {}
1:  procedure Strategy4(model, t1, t2, k)
2:      if plan is ∅ then
3:          ca_target ← computeCA(model, t2)
4:          seeds ← divide(ca_target, ⌈size(ca_target)/k⌉)
5:          for each seed ∈ seeds do
6:              ca ← computeCA(model, t1, seed)
7:              plan ← plan ∪ ca
8:      ca ← pick a ca ∈ plan
9:      plan ← plan − {ca}
10:     return ca
```

3.5. Strategy 5: Optimized coverage

The previous strategy (Section 3.4) guarantees to obtain a t_2-way coverage in a given number of days while achieving a t_1-way coverage on a daily basis (where $t_1 < t_2$). In doing so, however, it does not take the overhead cost into account. For this work, we define the *overhead cost* (in short, the *overhead*) as the number of additional configurations required (either on a daily basis or in total, depending on the context) to achieve a t_2-way coverage on top of a t_1-way coverage.

The optimized coverage strategy, on the other hand, takes as input a *daily overhead limit* and attempts to minimize the number of days required to obtain a t_2-way coverage while adhering to the given overhead limit and achieving a t_1-way coverage on a daily basis. The daily overhead for a given day is computed as the ratio of the size of the covering array required to be tested on the day by the CIT-daily process to the average size of the t_1-way covering arrays. For example, in a scenario where $t_1 = 2$ and $t_2 = 3$ with a daily overhead limit of 1.1, the optimized coverage strategy attempts to minimize the number of days required to achieve a 3-way coverage while guaranteeing a daily 2-way coverage, such that the number of additional configurations required to be tested on each day does not exceed the 10% of an average-sized 2-way covering array that can be created for the scenario.

As is the case with the previous strategy (Section 3.4), the optimized coverage strategy (Algorithm 6) leverages the seeding mechanism to guarantee a t_2-way coverage. In particular, this strategy uses the non-overlapping subsets of a t_2-way covering array as seeds to compute the t_1-way covering arrays to be used throughout the process. The sizes of the computed t_1-way

Algorithm 6 Optimized Coverage Strategy

```
    global plan ← {}
 1: procedure OptimizedCovStrategy(model, t1, t2, overhead, repeat)
 2:     if plan is ∅ then
 3:         plans_approx ← {}
 4:         size_base ← avgSize(model, t1)
 5:         ca_target ← computeCA(model, t2)
 6:         size_max ← ⌊∗⌋size_base ∗ overhead
 7:         ssize_min ← 1
 8:         ssize_max ← size_max
 9:         while ssize_min ≠ ssize_max do
10:             ssize ← ⌈(ssize_min + ssize_max)/2)⌉
11:             seeds ← divide(ca_target, ssize)
12:             plan′ ← {}
13:             for each seed ∈ seeds do
14:                 ca ← null
15:                 for repeat times do
16:                     ca′ ← computeCA(model, t1, seed)
17:                     if ca = null or |ca| > |ca′| then
18:                         ca ← ca′
19:                 plan′ ← plan′ ∪ ca
20:             if ∀ca ∈ plan′ → |ca| ≤ size_max then
21:                 plan ← plan′
22:                 ssize_min ← ssize
23:             else
24:                 plans_approx ← plans_approx ∪ plan′
25:                 ssize_max ← ssize
26:         if plan is ∅ then
27:             plan ← bestPlan(plans_approx)
28:     ca ← pick a ca ∈ plan
29:     plan ← plan − {ca}
30:     return ca
```

covering arrays should, however, be within the daily overhead limit.

One technical issue is that given the average size of a t_1-way covering array and the daily overhead limit, although it is straightforward to compute the maximum number of configurations that can be afforded on a single day, it is generally not possible to determine the size of a seed that can be used to compute a t_1-way covering array within the size limits. The optimized coverage strategy, therefore, employs a binary search-like algorithm to locate an appropriate plan. Note that binary search-like algorithms have been frequently used by covering array constructors to reduce the size of the covering arrays as much as possible (Mercan et al., 2018; Banbara et al., 2010). In this work, we, on the other hand, use them to determine the seed sizes in an attempt to keep the daily overheads within the allowed limits.

Note, however, that it may not always be possible to locate an appropriate plan where the daily overhead limit is satisfied each and every day. In such cases, the optimized coverage strategy locates an approximate plan by returning the plan with the lowest overall overhead among all the candidate plans identified during the search process. For this work, the *overall overhead* is computed as the difference between the total number of configurations required by the computed plan and those required by running an average-sized t_1-way covering array every day throughout duration of the plan.

Algorithm 6 presents the optimized coverage strategy. First, the average size of a t_1-way covering array is determined by constructing a number of covering arrays and computing their average size (line 4). Then, a t_2-way covering array (ca_{target}) is computed (line 5). Next, the maximum number of configurations that can be afforded each day ($size_{max}$) is determined (line 6). Finally, a binary search is carried out on the range of $[1, size_{max}]$ to determine the seed size (lines 9–25). To this end, the lower ($ssize_{min}$) and the upper bounds ($ssize_{max}$) for the seed size ($ssize$) are initially set to 1 (line 7) and $size_{max}$ (line 8), respectively.

For each seed size ($ssize$), ca_{target} is divided into non-overlapping seeds (line 11). For each seed, a number of t_1-way covering arrays are computed (line 16). Among all these covering arrays, the smallest one is chosen (lines 17–18) and included in the plan (line 19).

If the daily overhead limit is satisfied (line 20) for each and every day in the plan then a better plan (requiring a smaller number of days) is found (line 21). Thus, the lower bound ($ssize_{min}$) for the seed size is updated (line 22). Note that we seek to increase the seed size in an attempt to reduce the number of days required to achieve a full t_2-way coverage. Otherwise (line 23), an approximate plan is found. Therefore, the collection of approximate plans ($plans_{approx}$) as well as the upper bound for the seed size ($ssize_{max}$) is updated (lines 24 and 25, respectively).

At the end of the binary search, if an appropriate plan satisfying the daily overhead limit every day is found, then it is used. Otherwise (line 26), among all the approximate plans with the minimum overall cost, the shortest one is chosen and used as the plan (line 27). Either way, each covering array included in the selected plan is executed on a different day (lines 28–29). And, when there is no remaining covering arrays in the plan, a new plan is computed (lines 2–27).

4. Experiments

We have carried out a series of experiments, which are specifically designed to address our ultimate research question: How effective are the proposed strategies in terms of their fault revealing abilities? To this end, we have formulated the specific research questions given below. Note that, in all of these questions, the effectiveness of a strategy refers to the fault revealing ability of the strategy (Section 4.2).

1. RQ1: How effective is integrating fixed covering arrays into the daily build processes?
2. RQ2: How effective is varying the covering arrays to be tested throughout the daily build processes?
3. RQ3: How effective is increasing the coverage of higher strength tuples throughout the daily build processes?

4.1. Subjects

Subject applications. In the experiments, we used 3 well-known software projects, namely *JSPWiki* (Foundation, 2020c), *Cassandra* (Foundation, 2020a), and *Flink* (Foundation, 2020b) as our subject applications.

JSPWiki is a leading WikiWiki engine (Foundation, 2020c), Cassandra is a NoSQL distributed database management system (Foundation, 2020a), and Flink is a distributed processing engine for stateful computations over unbounded and bounded data streams (Foundation, 2020b). Further information about these subject applications can be found in Table 2.

We chose these subject applications to evaluate the proposed approach because (1) they are highly configurable software systems; (2) being open source projects, their source code repositories are accessible; (3) they are distributed with developer-created test suites; and (4) they possess the common characteristics of many configurable systems, which necessitate rigorous

Table 2
Subject applications used in the experiments.

SUD	Lines of code (≈ in millions)	Test cases
JSPWiki	0.22	1009
Cassandra	0.5	1872
Flink	1.8	8151

Table 3
Configuration space model used for JSPWiki.

Configuration options	Settings
diffProvider	{external, contextual, trad.}
encoding	{UTF-8, ISO-8859-1}
translatorReader.matchEnglishPlurals	{true, false}
urlConstructor	{shortView, short, default}
allowCreationOfEmptyPages	{true, false}
breakTitleWithSpaces	{true, false}
attachment.allowed	{png, jpg, zip, jar}
login.throttling	{true, false}
pageNameComparator.class	{human, local}
attachment.forbidden	{html, htm, php, asp, exe}
attachment.forceDownload	{html, htm}
searchProvider	{lucence, basic}
defaultprefs.template.editor	{plain, WikiWizard, FCK}
defaultprefs.template.sectionediting	{true, false}
defaultprefs.template.appearance	{true, false}
defaultprefs.template.autosuggest	{true, false}
defaultprefs.template.tabcompletion	{true, false}

Table 4
Configuration space model used for Cassandra.

Configuration options	Settings
authenticator	{allowAll, password}
authorizer	{allowAll, cassandra}
buffer.pool.use	{true, false}
trickle.fsync	{false, true}
rpc.keepalive	{true, false}
incremental.backups	{true, false}
endpoint.snitch	{simple, gossiping, property}
internode.compression	{dc, all, none}
inter.dc.tcp.nodelay	{false, true}
back.pressure.enabled	{false, true}
otc.coalescing.strategy	{disabled, fixed, movingaverage, timehorizon}

Table 5
Configuration space model used for Flink.

Configuration options	Settings
high-availability	{zookeeper, none}
high-availability.zookeeper	{creator, open, null}
web.submit.enable	{false, true}
classloader.resolve-order	{child-first, parent-first}
security.kerberos.login.use-ticket-cache	{false, true}
mesos.resourcemanager.ssl.enabled	{false, true}
mesos.resourcemanager.force-pull-image	{false, true}
state.backend.rocksdb.memory.managed	{false, true}
metrics.system-resource	{false, true}
state.backend.async	{false, true}
jobmanager.execution.failover-strategy	{full, region, null}
cluster.evenly-spread-out-slots	{false, true}
taskmanager.jvm-exit-on-oom	{false, true}
taskmanager.network.bind-policy	{ip, name}
security.ssl.rest.enabled	{false, true}
security.ssl.rest.authentication-enabled	{false, true}
security.ssl.internal.enabled	{false, true}
state.backend.incremental	{false, true}
state.backend	{jobmanager, filesystem, rocksdb}
yarn.per-job-cluster.include-user-jar	{first, last, ordered, disabled}

System-wide constraint:
High-availability = none $\implies$ high-availability.zookeeper = null

quality assurance and frequent integration, such as evolving continuously, having a large user as well as a code base, and being developed by a team of stakeholders.

Test suites. As the test suites in the experiments, we used the ones that came with the source code distributions of our subject applications (i.e., the ones that were developed by the developers of these applications). More specifically, we used a total of 1009, 1872, and 8151 test cases for JSPWiki, Cassandra, and Flink, respectively.

Configuration space models. To keep the cost of the experiments under control, we opted to use the configuration space models that are culled from the actual models. In particular, given the test suite of a subject application, we made sure that all the configuration options and their settings included in the respective model could potentially affect the behavior of the test cases. To this end, we read the user manuals of our subject applications, inspected their source codes, and ran small-scale experiments (as needed). Tables 3–5 present the configuration space models used in the experiments.

Time frames. For the experiments, the first question we faced was how to determine the duration of the experiments, i.e., the number of days, during which the CIT-daily process needed to be carried out. To be able to make fair comparisons between different CIT-daily strategies (Section 3), we opted to choose the durations, such that each strategy tested the same or similar number of configurations every day.

To this end, for a given subject application and a pair of coverage strengths (t_1 and t_2), we used the optimized coverage strategy (Section 3.5) to compute a plan with no overhead (i.e., with a daily overhead limit of 1.0, so that no additional configurations are required to be tested). We then used the number of days (d) needed by the plan as the duration of the respective experiments. That is, we let each CIT-daily strategy (starting from the same date) execute d consecutive days in the experiments. Note that the durations computed by the aforementioned approach, not only guarantee that all of the CIT-daily strategies are given enough time to achieve their coverage goals, but also ensure that these strategies test the same or similar number of configurations on a daily basis, which is the same as the average size of the t_1-way covering arrays that can be computed for the scenario at hand.

Once the duration d (in days) for an experiment was determined, the next question was to determine the start date for the experiment. To this end, for each subject application, we analyzed the source code repository of the application from 01/11/2019 to 30/06/2020 (i.e., for a period of 8 months) and located a d-day-long time frame, which represented a "typical" development period for the application in terms of the frequency of commits observed (i.e., the ratio of days with commits) between the aforementioned dates.

Table 6 presents the time frames used in the experiments. All of the CIT-daily strategies were carried out during the same time frames by using the same versions of the subject applications together with the same test suites.

4.2. Evaluation framework

To evaluate the proposed approach, we have used two metrics: *the number of missing t_2-tuples* for measuring the coverage of higher strength tuples and *the number of distinct failures* for measuring the fault revealing ability.

The former metric measures the total number of higher strength tuples (i.e., t_2-tuples where $t_2 = 3$ or 4) that are failed to be covered during the CIT-daily process. The lower the missing tuple count, the better the proposed approach is.

The latter metric, on the other hand, counts the total number of distinct SUD version, test case, and failure triplets observed,

Table 6
Time frames used in the experiments; the last two columns indicate the number of days the CIT-daily process was carried out and the number of days, in which at least one commit was experienced in the source code repository during this time frame, respectively.

Coverage strengths	SUD	Start date	End date	Duration (in days)	Days w/commits (versions)
$t_1 = 2$ $t_2 = 3$	JSPWiki	01/03/2020	06/03/2020	6	5
	Cassandra	17/02/2020	21/02/2020	5	5
	Flink	17/01/2020	21/01/2020	5	5
$t_1 = 2$ $t_2 = 4$	JSPWiki	01/03/2020	20/03/2020	20	14
	Cassandra	17/02/2020	27/02/2020	11	9
	Flink	17/01/2020	01/02/2020	16	16

where a triplet in this context consists of <SUD version, test case, failure>. We use the test oracles that come with the source code distribution of our subject applications to determine whether the test runs have failed or not. In the presence of a failure reported by a test oracle, we capture and parse the failure/error messages emitted by the SUD and/or the oracle to determine the distinct failures observed. Furthermore, to determine the SUD versions, we assign a version number to each SUD instance downloaded from the code repository. The version number of the SUD is considered to be 1 on the first day of the CIT-daily process. On each of the remaining days, if new commits (compared to the preceding day) are observed in the code repository, then the version number is incremented by one. Otherwise, the version number stays the same. Consequently, by counting the number of distinct SUD version, test case, and failure triplets, we aim to identify the different unexpected behaviors of the SUD.

Last but not least, all of the statistical significance tests are carried out by using the non-parametric Wilcoxon rank sum test (Haynes, 2013). Furthermore, a p-value less than 0.05 is considered to be statistically significant.

4.3. Operational framework

We experimented with CIT-daily scenarios, in which $t_1 = 2$ and $t_2 = 3$ as well as $t_1 = 2$ and $t_2 = 4$. In the remainder of the paper, a pair of t_1 and t_2 values together with a CIT-daily strategy evaluated on a particular SUD is referred to as an *experimental setup*.

As we have already discussed in Section 4.1, in order to be able to compare the effectiveness of different strategies in an unbiased manner, we executed all of the strategies exactly the same number of days without limiting their abilities, such that they test the same or similar number of configurations every day. From this perspective, the guaranteed coverage and the optimized coverage strategies become inseparable from each other; each strategy attempted to achieve a full t_2-way coverage in exactly the same number of days. Therefore, we carried a single set of experiments for the respective experimental setups. The results of these experiments are referred to as "guaranteed/optimized" in the remainder of the paper.

For each experimental setup, depending on the cost of the experiments, we repeated the experiments at least twice. In each repetition, we made sure that the strategy of interest generated a different sequence of covering arrays (e.g., a different plan).

To further increase the variations, we also created 50 simulated schedules for each actual repetition, in which we used the randomized, opportunistic, or guaranteed/optimized strategy. To this end, we replaced the covering arrays used by a strategy in a schedule with alternative covering arrays.

More specifically, let $[ca_1, ca_2, \ldots, ca_d]$ be a schedule of duration d days used by strategy s, where ca_i represents the covering array used for testing the version v_i of the system under development on day i ($1 \leq i \leq d$). Note that the version numbers start from 1 (i.e., $v_1 = 1$) and are incremented by one on each day where a change in the codebase occurs (Section 4.2). To replace the covering array ca_i in a given schedule, we first create the candidate list of covering arrays that can be used for the replacement. To this end, we determine all the other covering arrays tested on exactly the same version v_i of the system throughout the experiments regardless of the strategies, for which these covering arrays are used. Then, among all the covering arrays in the candidate list, we pick the one that best suits the strategy s. In particular, for the randomized coverage strategy, we randomly pick a covering array from the candidate list. For the opportunistic coverage strategy, we pick the covering array, which covers the most number of previously uncovered t_2-tuples up to the day i. And, for the guaranteed/opportunistic coverage strategy, we pick the covering array, which covers the most number of t_2-tuples that are not covered by the whole process (i.e., throughout the d days).

For each experimental setup, we also make sure that all of the generated schedules are distinct. In particular, given a schedule of duration d days, we randomly select some subsets of the days in the schedule, starting from the subsets of size one. Then, processing all possible subsets of the given size before increasing the size by one for the subsequent iteration until enough number of schedules (in our case, $cnt = 50$) are generated. For each selected subset, we follow the steps described above to replace each covering array in the subsequence with an alternative one, such that all of the generated alternative schedules are distinct.

Note that, in this approach, all of the covering arrays in the candidate list of a covering array ca_i are executed on exactly the same version v_i of the SUD with the covering array ca_i. Therefore, the test results obtained from each candidate covering array can safely be used to replace the ones obtained from ca_i in the original schedule. Note further that although this approach does not guarantee a full t_2-way coverage for the guaranteed/opportunistic coverage strategy, it created plans that were generally better than those created by the rest of the strategies in terms of the t_2-way coverage obtained (Table 8), which allowed us to use the respective plans in the analysis.

Moreover, to compare the effectiveness of the CIT-daily strategies to that of the common practices, we implemented a baseline strategy, called the *default coverage* strategy, which simply ran the test suites on the default configurations of the SUDs. That is, for a given SUD, this strategy, after downloading the latest version of the SUD, built the system without changing its default configuration, and then ran its test suite. Note that the first settings of the configuration options given in Tables 3–5 constitute the default configurations for our subject applications.

Throughout the experiments, we used the *surefire* plugin (v3.0.0.M4) with the *maven* (Foundation, 2020d) to configure and build the SUDs as well as to run the test cases and collect the results. Furthermore, the covering arrays used in the experiments were computed by Jenny – a well-known covering array constructor (Jenkins, 2005). We, in particular, chose this constructor due its ability to compute randomized covering arrays, which is well aligned with the ultimate goals of the proposed approach (Section 4.4). All the experiments were carried out on Google Cloud by using computing engines running Ubuntu 16.04 and 3.75 GB memory per vCPU.

Table 7
Summary statistics for the experiments where the columns depict the coverage strengths, SUDs, and the CIT-daily strategies used as well as the number of times the SUD was built, the number of distinct configurations tested, and the number of test runs carried, on average, respectively.

Coverage strengths	SUD	Strategy	Number of builds	Number of distinct configurations	Number of test runs
$t_1 = 2$ $t_2 = 3$	JSPWiki	default	6.00	1.00	6 004.00
		fixed	156.00	26.00	156 104.00
		randomized	156.33	156.33	156 437.33
		opportunistic	158.00	158.00	158 104.00
		guaranteed/optimized	149.00	149.00	149 098.67
	Cassandra	default	5.00	1.00	9 344.00
		fixed	85.00	17.00	158 811.33
		randomized	85.00	84.67	158 809.33
		opportunistic	85.00	85.00	158 807.00
		guaranteed/optimized	77.67	77.67	145 097.33
	Flink	default	5.00	1.00	39 504.00
		fixed	95.00	19.00	753 904.00
		randomized	95.00	95.00	756 507.67
		opportunistic	95.00	95.00	753 668.00
		guaranteed/optimized	90.50	90.50	724 085.00
$t_1 = 2$ $t_2 = 4$	JSPWiki	default	20.00	1.00	17 329.00
		fixed	520.00	26.00	450 554.00
		randomized	520.67	520.67	451 222.67
		opportunistic	525.00	525.00	455 570.50
		guaranteed/optimized	490.00	490.00	424 398.33
	Cassandra	default	11.00	1.00	20 572.00
		fixed	187.00	17.00	349 704.67
		randomized	187.00	185.33	349 722.67
		opportunistic	187.00	185.00	349 690.00
		guaranteed/optimized	181.33	180.00	339 097.00
	Flink	default	16.00	1.00	128 232.00
		fixed	304.00	19.00	2 412 102.00
		randomized	304.00	304.00	2 422 595.67
		opportunistic	304.00	304.00	2 418 670.00
		guaranteed/optimized	291.33	291.33	2 313 312.33

4.4. Data and analysis

All told, we, in total, built our subject applications 8674 times for 5736 distinct configurations and carried out more than 27 million test runs. Table 7 provides further statistics about these experiments. In this table, while the first three columns depict the coverage strengths, the subject applications, and the CIT-daily strategies we used in the experiments, respectively, the last three columns present the number of times the SUD was built, the number of distinct configurations used, and the number of test runs carried out by the respective CIT-daily process, on average.

We first observed that the covering array generator used in the experiments generated remarkably randomized covering arrays, which is well-aligned with the proposed approach as CIT-daily aims to vary the configurations tested as much as possible while maintaining a base coverage. In particular, for each of the randomized, opportunistic, and guaranteed/optimized coverage strategies, the sequence of covering arrays generated in a CIT-daily process barely shared any common configurations. This is, indeed, evident from the fact that the number of builds and the number of distinct configurations reported for these strategies in Table 7 are close to each other. Note that the number of builds is an upper bound for the number of distinct configurations. If no configuration is shared between the days, then the number of distinct configurations will be the same as the number of builds. The more the configurations are shared, the more the number of distinct configurations deviates.

We then observed that, as designed (Section 4.1), the CIT-daily strategies (except clearly for the default coverage strategy), used the same or similar number of configurations as well as test runs (Table 7), enabling an unbiased analysis of the results. Note that the slight variations in the number of test runs (when not caused by the differences in the number of configurations tested) are due to a well-known phenomenon; changing configurations may change the set of test cases that can run (Yilmaz, 2012).

We then analyzed the fault revealing abilities of the proposed approach. Fig. 1 and Table 8 summarize the results we obtained. For a given experimental setup, while the figure presents the distributions of the distinct failures observed, the table reports the average number of distinct failures as well as the coverage statistics for the higher strength tuples, i.e., t_2-tuples, namely the percentage of the t_2-tuples covered and the number of missing t_2-tuples, on average. In both cases, the entries superscripted with a '*' (star), indicate the average results obtained from both the actual and simulated repetitions, whereas the ones without any superscripts report the average results obtained from the actual repetitions only. In the remainder of the paper, since the results obtained from the actual and the simulated repetitions exhibit the same trend, we use the average results obtained from all the available repetitions.

Regarding RQ1: How effective is integrating fixed covering arrays into the daily build processes? Comparing the default coverage strategy to the fixed coverage strategy, we observed that the proposed approach significantly increased the fault revealing abilities of the daily build processes. In particular, when $t_1 = 2$ and $t_2 = 3$, the fixed coverage strategy increased the average number of distinct failures revealed from 10, 38, and 10 to 34, 87.33, and 55.67 for JSPWiki, Cassandra, and Flink, respectively. And, when $t_1 = 2$ and $t_2 = 4$, the numbers increased from 24, 71, and 35 to 96.33, 166, and 185.33.

RQ2: How effective is varying the covering arrays to be tested throughout the daily build processes? We then observed that varying the covering arrays to be tested throughout the process (rather than using a single covering array as is the case with the fixed coverage strategy) tended to increase the coverage of the higher strength tuples (Table 8). More specifically, when $t_1 = 2$ and $t_2 = 3$, the overall 3-tuple coverage percentages obtained

Table 8
The results obtained from different CIT-daily strategies where the columns depict the coverage strengths, SUDs, and CIT-daily strategies used as well as the percentage of the t_2-tuples (i.e., higher strength tuples) covered, the number of missing t_2-tuples observed, and the number of distinct failures revealed, on average, respectively.

Coverage strengths	SUD	Strategy	t_2-tuple coverage (%)	Missing t_2-tuple count	Number of distinct failures
$t_1 = 2$ $t_2 = 3$	JSPWiki	default	6.78	9 354.00	10.00
		fixed	82.45	1 761.00	34.00
		randomized	99.90	10.33	42.67
		opportunistic	99.94	6.50	50.00
		guaranteed/optimized	100.00	0.00	56.33
		randomized*	99.81	18.88	44.99
		opportunistic*	99.96	4.25	52.12
		guaranteed/optimized*	99.96	3.74	53.38
	Cassandra	default	7.74	1 967.00	38.00
		fixed	78.19	465.00	87.33
		randomized	99.70	6.33	92.33
		opportunistic	99.91	2.00	93.00
		guaranteed/optimized	100.00	0.00	105.67
		randomized*	99.41	12.50	90.40
		opportunistic*	99.91	1.98	92.67
		guaranteed/optimized*	99.94	1.28	102.94
	Flink	default	8.91	11 651.00	10.00
		fixed	85.20	1 893.67	55.67
		randomized	99.87	16.33	61.33
		opportunistic	99.90	13.00	65.50
		guaranteed/optimized	100.00	0.00	60.50
		randomized*	99.83	21.39	57.86
		opportunistic*	99.96	5.33	62.97
		guaranteed/optimized*	99.95	6.29	61.44
$t_1 = 2$ $t_2 = 4$	JSPWiki	default	2.80	82 595.00	24.00
		fixed	50.81	41 796.33	96.33
		randomized	99.92	72.00	119.33
		opportunistic	99.95	38.50	128.00
		guaranteed/optimized	100.00	0.00	139.00
		randomized*	99.80	169.10	108.80
		opportunistic*	99.97	23.28	121.35
		guaranteed/optimized*	99.98	12.96	136.88
	Cassandra	default	3.34	9 542.00	71.00
		fixed	45.53	5 377.67	166.00
		randomized	99.35	64.33	173.67
		opportunistic	99.64	36.00	176.00
		guaranteed/optimized	100.00	0.00	176.33
		randomized*	98.85	113.97	172.06
		opportunistic*	99.74	25.37	174.90
		guaranteed/optimized*	99.83	16.58	175.12
	Flink	default	4.02	115 677.00	35.00
		fixed	55.55	53 569.00	185.33
		randomized	99.93	86.00	187.00
		opportunistic	99.97	36.50	193.50
		guaranteed/optimized	100.00	0.00	195.00
		randomized*	99.86	165.20	190.12
		opportunistic*	99.97	30.56	191.50
		guaranteed/optimized*	99.99	14.03	194.56

by the fixed, randomized, and opportunistic coverage strategies were 81.95, 99.68, and 99.94, on average, respectively. And, when $t_1 = 2$ and $t_2 = 4$, the average 4-tuple coverage percentages were 50.63, 99.50, and 99.90.

Although the coverage statistics obtained from the randomized and opportunistic strategies were close to each other, the differences were nevertheless statistically significant in all of the 6 tests (3 SUDs x 2 pairs of coverage strengths). Note further that the guaranteed and optimized coverage strategies ensure a 100%t_2-tuple coverage. We, however, did not have the same guarantees in the simulated repetitions of these strategies (Section 4.3). But still, they provided generally better coverage statistics, compared to the randomized and opportunistic strategies. In particular, the average 3-tuple coverage percentage obtained by the guaranteed/optimized strategy was 99.95 when $t_1 = 2$ and $t_2 = 3$, and the average 4-tuple coverage percentage was 99.93 when $t_1 = 2$ and $t_2 = 4$ (Table 8). Furthermore, in 10 out of 12 tests carried out between the guaranteed/optimized strategy and the randomized as well as the opportunistic coverage strategy (3 SUDs x 2 pairs of coverage strengths x 2 pairs of strategies), the differences were statistically significant. We, therefore, opted to use the results obtained from these repetitions in the remainder of the paper.

We next identified a correlation (by using Pearson correlation coefficient) between the coverage of higher strength tuples (i.e., t_2-tuples) and the number of distinct failures revealed. Fig. 2 visualizes the results of the analysis. In particular, the higher the t_2-tuple coverage throughout the CIT-daily process, the more the distinct failures tended to be revealed.

Consequently, varying the covering arrays throughout the CIT-daily process, compared to using a fixed covering array, generally identified more distinct failures, on average (Fig. 1 and Table 8). While the average numbers of distinct failures revealed by the fixed coverage strategy were 34, 87.33, and 55.67 when

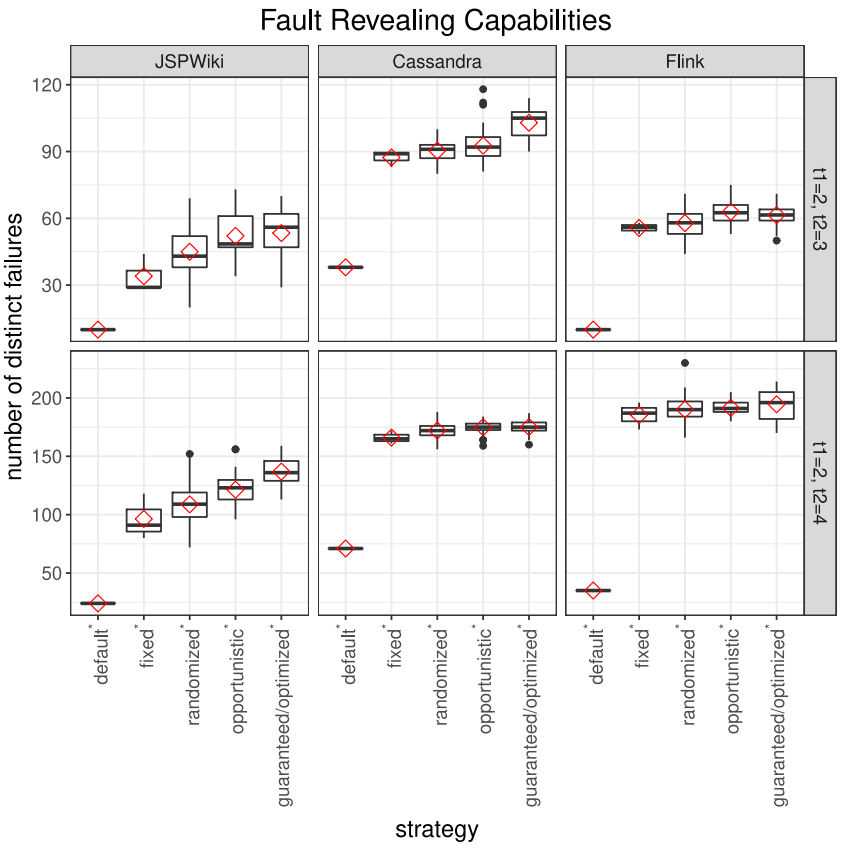

Fig. 1. The distributions of the number of distinct failures revealed by different CIT-daily strategies where the diamond symbols depict the average values.

$t_1 = 2$ and $t_2 = 3$ for JSPWiki, Cassandra, and Flink, respectively, those revealed by the randomized, opportunistic, and guaranteed/optimized coverage strategies, on average, were 50.16, 95.34, and 60.76. And, when $t_1 = 2$ and $t_2 = 4$, the numbers were 96.33, 166, and 185.33 vs. 122.34, 174.03, and 192.06.

RQ3: How effective is increasing the coverage of higher strength tuples throughout the daily build processes? As we go from the randomized coverage strategy to the opportunistic coverage strategy and then to the guaranteed/optimized coverage strategy (i.e., as the coverage of the higher strength tuples tended to increase), the averaged number of distinct failures increased, except for one case (Fig. 1 and Table 8). For example, when $t_1 = 2$ and $t_2 = 4$ for JSPWiki, the randomized, opportunistic, and guaranteed/optimized coverage strategies revealed 108.80, 121.35, and 136.88 distinct failures, on average, respectively. The only exception occurred when $t_1 = 2$ and $t_2 = 3$ for Flink, where the guaranteed/optimized strategy revealed slightly fewer distinct failures than the opportunistic strategy (61.44 vs. 62.97, on average). We believe that this singularity happened due to some intermittent failures.

Analyzing the statistical significance of these results, we observed that, in 7 out of 12 tests (3 SUDs $\times$ 2 pairs of coverage strengths $\times$ 2 pairs of strategies) made between the randomized and opportunistic coverage strategies as well as between the opportunistic and guaranteed/optimized coverage strategies, the differences were statistically significant. There was, however, no apparent trend. Comparing the guaranteed/optimized coverage strategy to the randomized coverage strategy (i.e., the two strategies at the opposite ends of the spectrum in terms of the higher strength tuple coverage), we, on the other hand, observed that in all of the 12 tests the differences were statistically significant.

Comparing the guaranteed coverage strategy with the optimized coverage strategy. As we have already discussed in Section 4.1, in order to be able to evaluate the proposed strategies, we needed to carry out them during the same period of days with the same or similar number of configurations tested on each day. Otherwise, it would have been quite difficult (if not impossible at all) to compare their fault revealing abilities in an unbiased manner. We have, therefore, set the parameters of these strategies accordingly in our experiments (Section 4.1), allowing us to compare the proposed strategies even with our baselines, including the fixed and randomized coverage strategies. This, however, made it difficult to understand the effects of the parameters in the guaranteed and optimized coverage strategies.

To this end, we have carried out an additional set of experiments. Note that the guaranteed and the optimized coverage strategies are designed to address different optimization concerns. More specifically, the guaranteed coverage strategy, by taking as input the number of days (k), in which the higher order coverage needs to be guaranteed, addresses the scenarios where meeting the deadlines has precedence over the overhead costs (Section 3.4). On the other hand, the optimized coverage strategy, which takes as input the daily overhead limit, addresses the scenarios where the overhead costs have precedence over meeting the deadlines (Section 3.5).

Both strategies, however, compute a plan that needs to be carried in a number of days. To reason about the effects of the aforementioned parameters, we, therefore, experimented with all possible durations from 2 days up to the durations reported in Table 6, which represent the durations for the daily overhead limit of 1.0. For each experimental setup, we, in particular, identified the parameter settings for the guaranteed and the optimized coverage strategies, which produced the same or similar plans in terms of both the duration and the number of configurations to be tested daily.

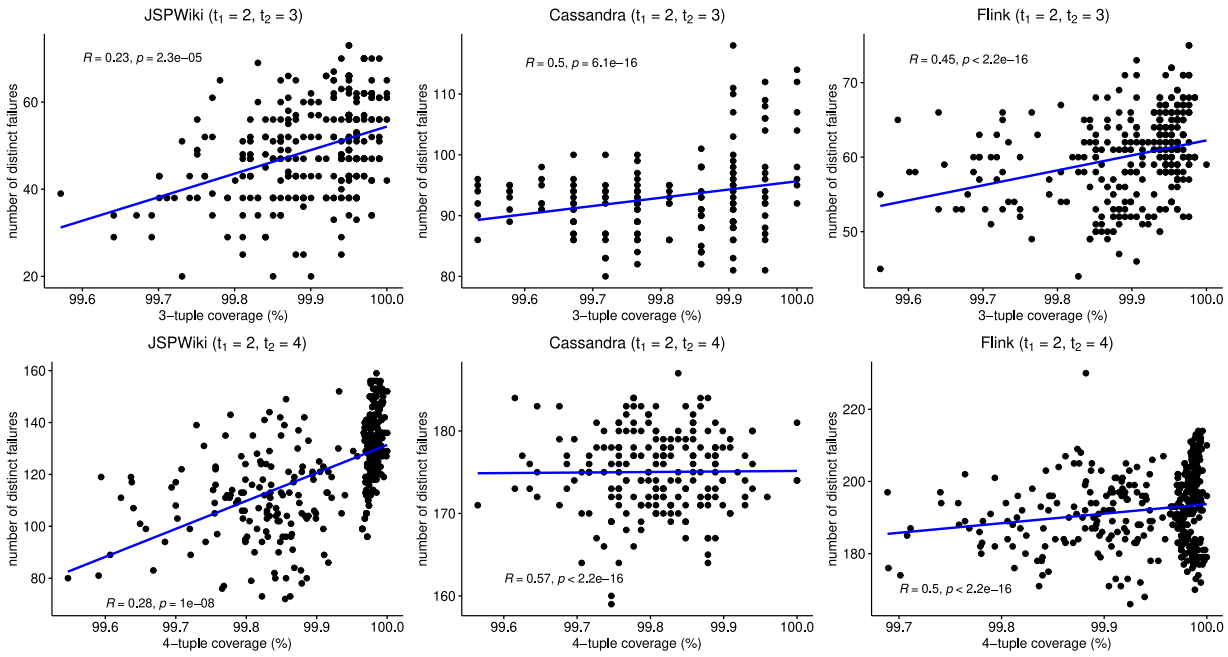

Fig. 2. The correlation between the coverage of higher strength tuples (i.e., t_2-tuples) and the number of distinct failures revealed. For a better visualization, only the coverage percentages, which are greater than 99.5, are plotted.

Table 9 presents the results we have obtained where the "duration in days (k)" and the "daily overhead limit" columns represent the input parameters of the guaranteed and the optimized coverage strategy, respectively. For each experimental setup, the experiments were repeated 5 times. As the average daily overhead limits increased, the durations decreased. And, as the durations increased, the daily overhead costs decreased. Furthermore, one interesting observation we make is that these tables, demonstrating the trade-off between the durations and the daily overhead costs, can actually be computed for the practitioners beforehand (i.e., before the CIT-daily process starts), so that educated decisions regarding the settings of the parameters can be made.

4.5. Discussion

Note that all of the CIT-daily strategies we have introduced in Section 3, use the existing covering array constructors to compute the test plans (i.e., sequence of covering arrays) to be used throughout the CIT-daily processes. As we discuss in detail in Section 6, this is a deliberate decision we make to be able to use the proposed strategies to integrate different types of CIT objects (not just the standard covering arrays) with the CIT-daily processes. Therefore, the scalability of these strategies is bounded by that of the existing constructors. That is, if the CIT constructor integrated with CIT-daily can compute the CIT objects (such as covering arrays) for a given test scenario, then CIT-daily will compute the test plan to be used in the daily build process.

Note further that the daily build processes are typically designed to be carried out in a single night, so that the results can be presented to the developers on a daily basis. To this end, the test cases to be used in the process are carefully selected/developed with their execution times in mind. When, using the CIT-daily process, on the other hand, the number of configurations to be tested, thus the number of test runs required, is dictated by the configuration space model used as well as the coverage criterion and the CIT-daily strategy employed. Consequently, the cost of carrying out all the required test runs on a single machine may require more than a night. For example, the daily execution times required for JSPWiki, Cassandra, and Flink were 4.2, 20.4, and 22.7 h, respectively. One way to resolve this issue to test the configurations included in a covering array in parallel. For example, had every configuration been tested on a different machine in our experiments, the processes would have taken 0.16, 1.20, and 1.19 h, on average, for JSPWiki, Cassandra, and Flink, respectively. Note that the configurations selected by a covering array are independent of each other. Therefore, testing configurations on different machines is a typically straightforward task, especially in the presence of virtual machines and cloud environments.

5. Threats to validity

One external threat to validity concerns the representativeness of the subject applications, the test suites, and the configuration space models used in the experiments. All the subject applications we used, however, were well-known, real-life applications, which possessed the common characteristics of many highly-configurable software systems (Section 4.1). Furthermore, the test suites used in the experiments were the ones that came with the source code distributions of our subject applications; they were all created by the actual developers of these applications. Last but not least, we developed the configuration space models, such that, given the test suite of a subject application, all the configuration options and their settings included in the model could potentially affect the behavior of the application. To this end, we read the user manuals, inspected the source codes, and ran small-scale experiments (as needed).

A related threat concerns the representativeness of the time frames, during which the CIT-daily processes were carried out in the experiments. We, however, determined the durations of the CIT-daily processes used in the experiments, such that all the strategies tested the same or similar number of configurations every day, which enabled us to compare the results obtained from different strategies in an unbiased manner (Section 4.1). Furthermore, to determine the start dates of the CIT-daily processes, we analyzed the source code repositories of our subject applications to find a period of time, which represented a typical development

Table 9
Comparing the guaranteed coverage strategy with the optimized coverage strategy where the "duration in days (k)" and the "daily overhead limit" columns represent the input parameters of the guaranteed and optimized coverage strategy, respectively. The daily overhead for a day is computed as the ratio of the size of the covering array required to be tested on the day to the average size of the t_1-way covering arrays.

Coverage strengths	SUD	Duration in days (k)	Daily overhead limit		
			min.	avg.	max.
$t_1 = 2$ $t_2 = 3$	JSPWiki	2	1.58	1.63	1.65
		3	1.08	1.15	1.27
		4	0.85	1.04	1.19
		5	0.85	0.96	1.08
	Cassandra	2	1.35	1.39	1.41
		3	0.94	1.07	1.18
		4	0.88	0.97	1.06
	Flink	2	1.47	1.57	1.63
		3	1.00	1.15	1.26
		4	0.89	1.01	1.16
$t_1 = 2$ $t_2 = 4$	JSPWiki	2	5.27	5.32	5.38
		3	3.46	3.58	3.65
		4	2.58	2.68	2.77
		5	2.12	2.15	2.23
		6	1.65	1.79	1.88
		7	1.42	1.59	1.69
		8	1.31	1.42	1.54
		9	1.19	1.32	1.46
		10	1.08	1.22	1.38
		11	0.96	1.18	1.31
		12	0.92	1.12	1.23
		13	0.88	1.09	1.27
		14	0.85	1.05	1.15
		15	0.85	1.04	1.23
		16	0.85	1.01	1.12
		17	0.85	1.00	1.12
		18	0.85	0.98	1.12
		19	0.85	0.98	1.08
	Cassandra	2	3.41	3.42	3.47
		3	2.18	2.30	2.35
		4	1.65	1.72	1.76
		5	1.24	1.44	1.53
		6	1.06	1.24	1.35
		7	1.00	1.14	1.35
		8	0.88	1.07	1.18
		9	0.88	1.04	1.18
		10	0.88	1.00	1.12
	Flink	2	4.47	4.62	4.74
		3	3.05	3.10	3.16
		4	2.16	2.27	2.32
		5	1.74	1.85	1.95
		6	1.42	1.59	1.68
		7	1.21	1.38	1.47
		8	1.16	1.29	1.42
		9	1.05	1.20	1.26
		10	0.95	1.13	1.26
		11	0.89	1.08	1.26
		12	0.95	1.05	1.16
		13	0.84	1.03	1.16
		14	0.84	1.00	1.16
		15	0.89	0.99	1.11

period for each application in terms of the frequency of commits observed (Section 4.1).

Another threat concerns the representativeness of the covering array constructor used in the experiments, namely Jenny (Jenkins, 2005). This constructor, however, is a well-known constructor, which has been used in many related works (Taş et al., 2017; Jia et al., 2015). Furthermore, we primarily chose this constructor due its ability to compute randomized covering arrays (Section 4.3), which is well aligned with the ultimate goals of the CIT-daily approach (Section 4.4). Nevertheless, Jenny has some known limitations, including the limit on the maximum number of values, which can be assumed by parameters (Mercan et al., 2022).

6. Related work

Daily build processes are generally considered to be a part of the continuous integration activities, commonly referred to as CI (Fowler and Foemmel, 2006; Ståhl and Bosch, 2014). CI is a software development practice where members of a development team integrate and test their works frequently, so that integration errors can be detected as quickly as possible, which in turn can decrease the turnaround time for fixes. Consequently, the daily build processes have been studied in many domains including the open source projects (Holck et al., 2003; Ingo and Daly, 2020), distributed systems (Karlsson et al., 2000; Koroorian and Kajko-Mattsson, 2008), and graphical user interfaces (Memon et al., 2003). In this work, we enhance the daily build processes with combinatorial interaction testing.

One common issue in CI is to figure out how frequently the system under development needs to be integrated and tested (Felidré et al., 2019; Ståhl and Bosch, 2014). Some of the studies in the literature argue that the CI activities need to be carried out after every commit to the codebase (Woskowski, 2012), whereas others propose to run these activities regularly at predefined intervals (Virmani, 2015; Holck et al., 2003), such as once on a daily basis as is the case with the daily build processes (Holck et al., 2003), multiple times per day (Stolberg, 2009), or once every few hours (Rogers, 2004). It is also possible to develop hybrid CI approaches where certain activities are performed on a fixed schedule, while others are triggered as needed by the source code changes (Woskowski, 2012; Kowzan and Pietrzak, 2019). Furthermore, not only the frequency, but also the duration of the CI activities was a subject of research (Woskowski, 2012). Note that CIT-daily can readily be modified to carry out the requested CI activities at any frequency or at will as long as the durations/costs of the activities are kept under control (in particular, the duration/cost of testing a single configuration). To this end, the fact that multiple configurations can be tested in parallel (Section 4.5), can further help the practitioners control the durations.

Combinatorial interaction testing (CIT) aims to efficiently and effectively reveal the failures that are caused by the interactions of different factors/parameters affecting the system's behavior (Yilmaz et al., 2013b; Nie and Leung, 2011). Since CIT can generally be applicable as long as the system under test has more than one interacting factor/parameter, it has been extensively used for testing software systems in a wide spectrum of domains (Jarman and Smith, 2019; Rao and Li, 2021; Yilmaz et al., 2006; Qian et al., 2018; Michaels et al., 2020; Qi et al., 2018; Choi, 2017).

One of the well-known and frequently used CIT objects for testing is *covering arrays*, providing t-way coverage guarantees (Yu et al., 2013; Yilmaz et al., 2013b). Different approaches can be used to construct the standard covering arrays, including mathematical approaches (Sarkar et al., 2018; Kampel et al., 2018), greedy approaches (Lei et al., 2008b; Wang and He, 2013), and search-based metaheuristic approaches (Mercan et al., 2018; Jia et al., 2015).

Other types of CIT objects include the cost-aware covering arrays (Demiroz and Yilmaz, 2012) and the test case-aware covering arrays (Yilmaz, 2012). Given a configuration space model and a cost function modeling the actual cost of testing, a t-way cost-aware covering array is a t-way covering array that "minimizes" the given cost function, which is not necessarily the same thing as reducing the number of configurations required when the cost of testing varies from one configuration to another. And, the test case-aware covering arrays (Yilmaz, 2012) take the

test case-specific constraints into account when constructing the covering arrays, such that, for each test case, every valid t-tuple is covered at least once, converting the covering array from a list of configurations to a list of configurations, each of which is associated with a set of test cases scheduled to be executed on the configuration. Note that, in the presence of test case-specific constraints, the set of valid t-tuples to be covered may vary from one test case to another.

Moreover, Unified Combinatorial Interaction Testing (U-CIT) is a novel approach introduced recently (Mercan et al., 2022; Mercan and Yilmaz, 2016), which aims to improve the applicability of CIT in practice by providing a unified way of computing the CIT objects, so that domain-, application-, or project-specific CIT objects for testing can be defined and computed even by the practitioners. What makes U-CIT a unified approach is that both the entities to be covered and the space of test cases, from which the samples will be drawn, are expressed as constraints, which turns the construction of the CIT objects into an interesting constraint solving problem, called *cov-CSP* (Mercan and Yilmaz, 2016).

Perhaps, the closest work to ours is the one presented in Segall (2016). In the aforementioned work, Segall introduces three algorithms for repeated combinatorial test design, namely the random algorithm, the generic algorithm, and a simulated annealing-based algorithm. The first two algorithms correspond to the randomized and the opportunistic coverage strategies we discussed in Section 3, respectively, both of which we, indeed, use as baselines. In the last algorithm, Segall implements a specialized constructor by extending CASA (a simulated annealing-based covering array constructor (Garvin et al., 2011)) to compute a sequence of t-way covering arrays for $(t + 1)$-way coverage. To this end, the energy function of the simulated annealing algorithm is modified, such that the neighboring states covering more of the previously uncovered $(t + 1)$-tuples become more preferable.

In some of our recent works, we have been, indeed, extensively arguing that the need for developing specialized CIT constructors profoundly restricts the applicability of CIT in practice (Mercan et al., 2022; Mercan and Yilmaz, 2021, 2016). In this work, we, therefore, developed a number of CIT-daily strategies, especially the guaranteed and the optimized coverage strategies, which deliberately treat the underlying CIT constructors as black boxes. More specifically, our CIT-daily strategies can readily be used with any CIT constructor that supports a seeding mechanism and that can randomize the generation of the CIT objects (Section 3). Therefore, our strategies readily allow not only the standard covering arrays (as was the case in this work), but also the other types of CIT objects, including the cost-aware covering arrays (Demiroz and Yilmaz, 2012), the test case-aware covering arrays (Yilmaz, 2012), and the U-CIT objects (Mercan et al., 2022), to be integrated with the daily build processes. If specialized constructors were to be used instead, then for each specific type of CIT object to be integrated with a daily build process, a new constructor may need to be developed, which would profoundly restrict the applicability of CIT-daily in practice.

Furthermore, for a given strength t, the only parameter that the practitioners can use in the aforementioned CASA-based constructor is the α parameter, which controls the trade-off between the minimality of the t-way covering arrays and their effectiveness in covering $(t+1)$-way tuples. Expressing practical resource constraints by using this parameter, such as the duration limits as we do with the guaranteed coverage strategy or the daily overhead limits as we do with the optimized coverage strategy, can, however, be difficult.

Last but not least, another way our work differs from Segall's work is that while Segall is focusing on the convergence rates of the $(t + 1)$-way coverage provided by different algorithms, we focus on the actual fault revealing abilities of our CIT-daily strategies by empirically evaluating them on real and evolving software systems.

7. Concluding remarks and future works

In this work, we have introduced CIT-daily, which integrates combinatorial interaction testing approaches with the daily build processes to systematically test the interactions between the factors/parameters (such as, configuration options) that can affect the behavior of the systems, on a daily basis.

We have, furthermore, developed a number of CIT-daily strategies, namely fixed, randomized, opportunistic, guaranteed, and optimized coverage strategies. While the fixed coverage strategy uses the same t-way covering array throughout the whole process, the randomized coverage strategy attempts to test a different t-way covering array every day with the goal of randomizing the coverage of higher strength tuples while guaranteeing a daily t-way coverage. The opportunistic coverage strategy, on the other hand, takes as input two coverage strengths, namely t_1 and t_2, and aims to opportunistically maximize the coverage of higher strength tuples, i.e., t_2-tuples where $t_2 > t_1$, throughout the process while guaranteeing a t_1-way coverage every day. Last but not least, the guaranteed and optimized coverage strategies, while achieving a daily t_1-way coverage, guarantee a t_2-way coverage for a given number of days (as is the case with the guaranteed coverage strategy) or for a given daily overhead limit (as is the case with the optimized coverage strategy).

We have empirically evaluated the effectiveness (i.e., fault revealing abilities) of the proposed approach on real-life, highly configurable software systems. We observed that (1) the proposed approach significantly improved the effectiveness of the daily build processes, compared to the current daily build practices where all the test cases are executed on the default configurations of the systems; (2) randomizing the coverage of higher strength combinations (i.e., t_2-way combinations) while guaranteeing a base t_1-way coverage on a daily basis, further improved the effectiveness; and (3) the more the higher strength combinations were covered during the process, the higher the fault revealing abilities tended to be.

One avenue for future work is to develop feedback-driven CIT-daily processes (inspired from (Dumlu et al., 2011; Yilmaz et al., 2013a)) where the failures observed during the process can be used to decide what needs to be tested on the succeeding days. Such approaches, by avoiding the harmful consequences of masking effects (Dumlu et al., 2011; Yilmaz et al., 2013a), can further improve the effectiveness of CIT-daily. Similarly, the generated covering arrays for a CIT-daily process can also be post-processed to remove the configurations, which are not actually contributing to the coverage properties, or to replace the same configurations appearing on multiple days by distinct configurations without affecting the coverage properties of the entire CIT-daily process to further randomize the configurations to be tested.

CRediT authorship contribution statement

Hanefi Mercan: Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing. **Atakan Aytar:** Software, Data Curation. **Giray Coskun:** Software, Data Curation. **Dilara Mustecep:** Software, Resources. **Gülsüm Uzer:** Data curation. **Cemal Yilmaz:** Conceptualization, Supervision, Formal analysis, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Alazzawi, A.K., Rais, H.M.e.a., 2019. Phabc: A hybrid artificial bee colony strategy for pairwise test suite generation with constraints support. In: 2019 IEEE Student Conference on Research and Development (SCOReD). pp. 106–111. http://dx.doi.org/10.1109/SCORED.2019.8896324.

Banbara, M., Matsunaka, H., Tamura, N., Inoue, K., 2010. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In: Logic for Programming, Artificial Intelligence, and Reasoning. Springer pp. 112–126.

Charbachi, P., Eklund, L., Enoiu, E., 2017. Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers? In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 92–99. http://dx.doi.org/10.1109/QRS-C.2017.23.

Choi, J.S., 2017. Controller-centric combinatorial wrap-around interaction testing to evaluate a stateful pce-based transport network architecture. IEEE/OSA J. Opt. Commun. Networking 9, 792–802. http://dx.doi.org/10.1364/JOCN.9.000792.

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23, 437–444.

Demiroz, G., Yilmaz, C., 2012. Cost-aware combinatorial interaction testing. In: Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles. pp. 9–16.

Dumlu, E., Yilmaz, C., Cohen, M.B., Porter, A., 2011. Feedback driven adaptive combinatorial testing. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 243–253.

Felidré, L., da Costa, D.A., Cartaxo, B., Pinto, G., 2019. Continuous integration theater. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 1–10.

Fouché, M.B., Porter, A., 2009. Incremental covering array failure characterization in large configuration spaces. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 177–188.

Foundation, A.S., 2020a. Apache cassandra. URL: https://cassandra.apache.org/.

Foundation, A.S., 2020b. Apache flink. URL: https://flink.apache.org/.

Foundation, A.S., 2020c. Apache jspwiki. URL: https://jspwiki.apache.org/.

Foundation, A.S., 2020d. Maven surefire plugin. URL: https:https://maven.apache.org/surefire/maven-surefire-plugin/.

Fowler, M., Foemmel, M., 2006. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html.

Galinier, P., Kpodjedo, S., Antoniol, G., 2017. A penalty-based Tabu search for constrained covering arrays. In: Proceedings of the Genetic and Evolutionary Computation Conference. ACM, pp. 1288–1294.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. 16, 61–102.

Gladisch, C., Heinzemann, C., Herrmann, M., Woehrle, M., 2020. Leveraging combinatorial testing for safety-critical computer vision datasets. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. pp. 324–325.

Hartman, A., 2005. Software and hardware testing using combinatorial covering suites. In: Graph Theory, Combinatorics and Algorithms. Springer pp. 237–266.

Haynes, W., 2013. Wilcoxon Rank Sum Test. Springer New York, pp. 2354–2355. http://dx.doi.org/10.1007/978-1-4419-9863-7_1185.

Holck, J., Jørgensen, N., et al., 2003. Continuous integration and quality assurance: A case study of two open source projects. Aust. J. Inf. Syst. 11.

Ingo, H., Daly, D., 2020. Automated system performance testing at mongodb. In: Proceedings of the Workshop on Testing Database Systems. pp. 1–6.

Jarman, D., Smith, R.e.a, 2019. Applying combinatorial testing to large-scale data processing at adobe. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 190–193.

Jenkins, B., 2005. jenny: A pairwise testing tool. http://www.burtleburtle.net/bob/index.html.

Jia, Y., Cohen, M.B., Harman, M., Petke, J., 2015. Learning combinatorial interaction test generation strategies using hyperheuristic search. In: Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE). IEEE, pp. 540–550.

Kampel, L., Garn, B., Simos, D.E., 2018. Covering arrays via set covers. Electron. Notes Discrete Math. 65, 11–16.

Karlsson, E.A., Andersson, L.G., Leion, P., 2000. Daily build and feature development in large distributed projects. In: Proceedings of the 22nd International Conference on Software Engineering. pp. 649–658.

Koroorian, S., Kajko-Mattsson, M., 2008. A tale of two daily build projects. In: 2008 the Third International Conference on Software Engineering Advances. IEEE, pp. 245–251.

Kowzan, M., Pietrzak, P., 2019. Continuous integration in validation of modern, complex, embedded systems. In: 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP). IEEE, pp. 160–164.

Kuhn, D.R., Kacker, R.N., Lei, Y., et al., 2010. Practical combinatorial testing. NIST Spec. Publ. 800, 142.

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

Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2008b. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verif. Reliab. 18, 125–148.

Li, Z., Chen, Y., Gong, e.a, 2019. A survey of the application of combinatorial testing. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 512–513. http://dx.doi.org/10.1109/QRS-C.2019.00100.

Lopez-Herrejon, R.E., Fischer, S., Ramler, R., Egyed, A., 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10. http://dx.doi.org/10.1109/ICSTW.2015.7107435.

Ma, L., Zhang, F., Xue, M., Li, B., Liu, Y., Zhao, J., Wang, Y., 2018. Combinatorial testing for deep learning systems. arXiv preprint arXiv:1806.07723.

Memon, A., Banerjee, I., Hashmi, N., Nagarajan, A., 2003. Dart: a framework for regression testing nightly/daily builds of gui applications. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. IEEE, pp. 410–419.

Mercan, H., Javeed, A., Yilmaz, C., 2022. Flexible combinatorial interaction testing. IEEE Trans. Softw. Eng. 48, 1030–1066. http://dx.doi.org/10.1109/TSE.2020.3010317.

Mercan, H., Yilmaz, C., 2016. A constraint solving problem towards unified combinatorial interaction testing. In: Proceedings of the 7th Workshop on Constraint Solvers in Testing, Verification, and Analysis. CEUR, pp. 24–30.

Mercan, H., Yilmaz, C., 2021. Computing sequence covering arrays using unified combinatorial interaction testing. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE pp. 77–80.

Mercan, H., Yilmaz, C., Kaya, K., 2018. Chip: A configurable hybrid parallel covering array constructor. IEEE Trans. Softw. Eng. 45, 1270–1291.

Michaels, R., Adamo, D., Bryce, R., 2020. Combinatorial-based event sequences for reduction of android test suites. In: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). pp. 0598–0605. http://dx.doi.org/10.1109/CCWC47524.2020.9031238.

Mirzaei, N., Garcia, J.e.a, 2016. Reducing combinatorics in gui testing of android applications. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 559–570. http://dx.doi.org/10.1145/2884781.2884853.

Mohammad, S.A.K., Valepe, S.V.e.a, 2019. A comparative study of the effectiveness of meta-heuristic techniques in pairwise testing. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). pp. 91–96. http://dx.doi.org/10.1109/COMPSAC.2019.00022.

Mukelabai, M., Nešić, D.e.a, 2018. Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 155–166. http://dx.doi.org/10.1145/3238147.3238201.

Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Comput. Surv. 43, 1–29.

Qi, G., Tsai, W.T., Colbourn, C.J., Luo, J., Zhu, Z., 2018. Test-algebra-based fault location analysis for the concurrent combinatorial testing. IEEE Trans. Reliab. 67, 802–831. http://dx.doi.org/10.1109/TR.2018.2833449.

Qian, Y., Zhang, C., Wang, F., 2018. Selecting products for high-strength t-wise testing of software product line by multi-objective method. In: 2018 IEEE International Conference on Progress in Informatics and Computing (PIC). pp. 370–378. http://dx.doi.org/10.1109/PIC.2018.8706270.

Rao, C., Li, N.e.a, 2021. Combinatorial test generation for multiple input models with shared parameters. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3065950.

Rogers, R.O., 2004. Scaling continuous integration. In: Extreme Programming and Agile Processes in Software Engineering. Springer Berlin Heidelberg pp. 68–76.

Sarkar, K., Colbourn, C.J., De Bonis, A., Vaccaro, U., 2018. Partial covering arrays: algorithms and asymptotics. Theory Comput. Syst. 62, 1470–1489.

Segall, I., 2016. Repeated combinatorial test design–Unleashing the potential in multiple testing iterations. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 12–21.

Ståhl, D., Bosch, J., 2014. Modeling continuous integration practice differences in industry software development. J. Syst. Softw. 87, 48–59.
Stolberg, S., 2009. Enabling agile testing through continuous integration. In: 2009 Agile Conference. pp. 369–374. http://dx.doi.org/10.1109/AGILE.2009.16.
Taş, M.K., Mercan, H., Demiröz, G., Kaya, K., Yilmaz, C., 2017. Generating cost-aware covering arrays for free. In: International Conference on Tools and Methods for Program Analysis. Springer, pp. 170–182.
Virmani, M., 2015. Understanding devops & bridging the gap from continuous integration to continuous delivery. In: Fifth International Conference on the Innovative Computing Technology (Intech 2015). IEEE, pp. 78–82.
Wang, Z., He, H., 2013. Generating variable strength covering array for combinatorial software testing with greedy strategy. JSW 8, 3173–3181.
Woskowski, C., 2012. Applying industrial-strength testing techniques to critical care medical equipment. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 62–73.
Yilmaz, C., 2012. Test case-aware combinatorial interaction testing. IEEE Trans. Softw. Eng. 39, 684–706.
Yilmaz, C., Cohen, M.B., Porter, 2006. Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans. Softw. Eng. 32, 20–34.
Yilmaz, C., Dumlu, E., Cohen, M.B., Porter, A., 2013a. Reducing masking effects in combinatorial interaction testing: A feedback drivenadaptive approach. IEEE Trans. Softw. Eng. 40, 43–66.
Yilmaz, C., Fouche, S., Cohen, M.B., Porter, A., Demiroz, G., Koc, U., 2013b. Moving forward with combinatorial interaction testing. Computer 47, 37–45.
Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R., 2013. Acts: A combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, pp. 370–375.

Hanefi Mercan received the BS degree in mathematics in 2012. He received MS and Ph.D. degrees in computer science and engineering in 2015 and 2021, respectively. He is currently working as a Data Scientist expert at Turk Telekom, Istanbul, Turkey.

Atakan Aytar received his BS degree in computer science and engineering from Sabanci University, Istanbul, Turkey, in 2021.

Giray Coskun is currently an undergraduate student at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey.

Dilara Mustecep is currently an undergraduate student at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey.

Gülsüm Uzer received her BS and MS degrees in computer science and engineering from Bilkent University, Ankara, Turkey, in 2017, and from Sabanci University, Istanbul, Turkey, in 2019, respectively.

Cemal Yilmaz received the BS and MS degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1997 and 1999, respectively. In 2005, he received the Ph.D. degree in computer science from the University of Maryland at College Park. Between 2005 and 2008, he worked as a post-doctoral researcher at IBM Thomas J. Watson Research Center, Hawthorne, New York. He is currently an associate professor of computer science in the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. His current research interests include Software Engineering, Software Quality Assurance, and Software Security.