Postprint of article in *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 9, No. 2, Article 9, 2014

Improving the Effectiveness of Testing Pervasive Software via Context Diversity

HUAI WANG, The University of Hong Kong
W. K. CHAN, City University of Hong Kong
T. H. TSE, The University of Hong Kong

Context-aware pervasive software is responsive to various contexts and their changes. A faulty implementation of the context-aware features may lead to unpredictable behavior with adverse effects. In software testing, one of the most important research issues is to determine the sufficiency of a test suite to verify the software under test. Existing adequacy criteria for testing traditional software, however, have not explored the dimension of serial test inputs and have not considered context changes when constructing test suites. In this article, we define the concept of *context diversity* to capture the extent of context changes in serial inputs and propose three strategies to study how context diversity may improve the effectiveness of the data-flow testing criteria. Our case study shows that the strategy that uses test cases with higher context diversity can significantly improve the effectiveness of existing data-flow testing criteria for context-aware pervasive software. In addition, test suites with higher context diversity are found to execute significantly longer paths, which may provide a clue that reveals why context diversity can contribute to the improvement of effectiveness of test suites.

Categories and Subject Descriptors: **D.2.5 [Software Engineering]**: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms: Verification, Experimentation, Measurement, Reliability

Additional Key Words and Phrases: Context-aware program, context diversity, test adequacy

ACM Reference Format:

Huai Wang, W. K. Chan, and T. H. Tse. 2014. Improving the effectiveness of testing pervasive software via context diversity. *ACM Trans. Autonom. Adapt. Syst.* 9, 2, Article 9 (June 2014), 28 pages. DOI: http://dx.doi.org/10.1145/2620000

1. INTRODUCTION

Context-aware pervasive software (CPS) applications capture the evolution of the computing environment as *contexts*, and self-adapt their behavior dynamically according to such contexts [Lu et al. 2008]. For example, a smartphone equipped with sensors may continuously sample the user's contexts such as locations and activities and use them as further inputs to switch among its modes: when receiving a coming

This research is supported in part by the Early Career Scheme and the General Research Fund of the Research Grants Council of Hong Kong (project numbers 123512, 717811, and 716612).

Authors' addresses: Huai Wang and T. H. Tse, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong; emails: {hwang, thtse}@cs.hku.hk; W. K. Chan (corresponding author), Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong; email: wkchan@cityu.edu.hk.

DOI:http://dx.doi.org/10.1145/2620000

call, it may vibrate silently when the user is presenting a report in a meeting room or may beep loudly when the user is watching a football game at home.

This class of applications is increasingly deployed in our everyday environments. At the same time, their failures, if any, increasingly affect our daily living. It is critical to ensure their quality.

Software testing is the most widely used approach in the industry to ensure the quality of programs [Bertolino 2007]. In the testing process, test inputs known as *test cases* are applied to the program under test to verify whether the execution results agree with the expected outcomes. Any disagreement between the two shows that the execution of the test case against the program reveals a *failure* of the program under test. Every such failure is due to a *fault*, which means an incorrect step in a program.

However, it is well known that testing techniques can only reveal failures but cannot prove the absence of faults [Ammann and Offutt 2008], and it is impractical to exhaustively test the program with nontrivial input domains [Weyuker and Ostrand 1980]. As such, a central problem in software testing research is to determine when to stop the testing process, which gives birth to research in test adequacy criteria [Hutchins et al. 1994]. Specifically, a collection of test cases, known as a *test suite*, is said to be *adequate* if it satisfies a specific test adequacy criterion. A simple test adequacy criterion is statement coverage [Zhu et al. 1997], where every statement in the code is executed at least once by the test suite. More advanced test adequacy criteria have been introduced in the literature. An important class of such criteria is the *data-flow testing criteria*, which examine how well test cases cover the execution paths between data definitions and their corresponding data usage in the program under test [Frankl and Weyuker 1988]. Data-flow testing criteria have been widely regarded to be highly effective, which has been validated by numerous empirical studies [Hutchins et al. 1994; Frankl and Weiss 1993; Offutt et al. 1996].

Traditional test adequacy criteria may, however, fail to perform effectively on some classes of applications due to domain-specific features. As such, domain-specific data-flow testing criteria have been proposed to address this problem. Examples include test adequacy criteria for database-driven applications [Kapfhammer and Soffa, 2003], service-oriented workflow applications [Mei et al. 2008], and CPS applications [Lai et al. 2008; Lu et al. 2006, 2008] to address the challenges due to embedded SQL statements, XML and XPath constructs, and dynamic evolutions of contexts in the program environment, respectively.

In general, test cases for CPS applications consist of sequences of context values. For instance, the smartphone takes ⟨*location*, *activity*⟩ sequences such as ⟨meeting room, present report⟩ and ⟨home, watch football⟩ as inputs. Context values are usually noisy and error prone [Chen et al. 2011]. CPS applications running with reference to these context values are, however, expected to compute results that agree with the users' intuitions on CPS applications perceived from the surrounding environment.

Our experience in conducting the experimentation for the aforementioned criteria for CPS applications [Lu et al. 2006] shows that CPS programs over different context sequences serving as inputs may follow the same program path, and yet the fault detection ability of these context sequences may be significantly different. This leads us to develop our work in Wang and Chan [2009], which proposes the notion of *context diversity* to measure the extent of context changes inherent in sequences of context values. For example, the context diversity for the sequences ⟨meeting room, present report⟩ and ⟨home, watch football⟩ is 2 after summing up the context changes in the dimensions of location (where “meeting room” is different from “home”) and activity (where “present report” is different from “watch football”).

This article extends our previous work [Wang and Chan 2009] to study how context diversity affects the effectiveness of data-flow testing criteria in exposing faults in CPS applications. Specifically, we study three strategies, namely, *CARS-H*, *CARS-L*, and *CARS-E*, to select test cases with higher, lower, and more evenly distributed context diversity, respectively, in constructing test suites that are adequate with respect to the data-flow testing criteria. To study these effects, we report a new multisubject case study using the popular and representative data-flow testing criteria on the three CPS benchmarks with a total of 8,097 lines of code, 30,000 test cases, 959 faulty versions produced through mutation analysis [Andrews et al. 2006; Budd et al. 1980], and 43,200 adequate test suites. Our case study covers all the benchmarks used in representative previous work such as Lu et al. [2008], Wang et al. [2007], and Zhai et al. [2010, 2014].

The experimental results in the case study show that *CARS-H* and *CARS-E* can improve the effectiveness of data-flow testing criteria by 10.6%−22.1% and 0.8%−5.3%, respectively, while *CARS-L* can be 2.0%−22.2% less effective. Considering that data-flow testing criteria have been deemed by many researchers to be highly effective test adequacy criteria to expose faults, the experimental results indicate that the additional boost by *CARS-H* in terms of test effectiveness is significant. The contrast between *CARS-H* and *CARS-L* further confirms that the difference is not by chance. Moreover, the ineffectiveness of evenly distributed test case diversity (demonstrated by *CARS-E*) in the CPS domain points out potential further research in random testing and adaptive random testing.

The main contribution of this article is threefold: (i) it is the first work to formulate strategies to modify the context-awareness distribution of adequate test suites, (ii) we report one of the largest case studies on the effectiveness and ineffectiveness of testing strategies for CPS applications, and (iii) we propose the notion of context diversity and provide the first empirical evidence to demonstrate its usefulness in enhancing the effectiveness of testing CPS applications.

The rest of the article is organized as follows: Section 2 reviews related work. Section 3 introduces the fundamental concepts in this article and formulates our test suite construction strategies. Section 4 presents the research questions and explains the setup of the case study. Section 5 summarizes the analysis results. Section 6 discusses the applicability of traditional data-flow adequacy criteria to pervasive software, and Section 7 concludes the article.

2. RELATED WORK

Many researchers proposed various verification techniques and methodologies to ensure the quality of CPS applications. Tse et al. [2004] advocated the use of metamorphic relations among different contexts to alleviate the test oracle problem. Lu et al. [2006, 2008] identified new data-flow associations that projected the effects of context changes on the traditional control-flow graphs of CPS applications, and proposed a family of test adequacy criteria. Wang et al. [2007] proposed to manipulate the interleaving of multithreaded components with respect to a set of program call sites where context changes may affect the program states, and further recommended a set of control-flow testing criteria to exercise all these program points. Lai et al. [2008] proposed a set of coverage-based test adequacy criteria to expose interrupt-based faults in nesC programs while the programs readjusted their behavior to new contexts. Our technique proposed in the present article can be integrated with these coverage-based criteria by selecting test cases with different context diversity optimization objectives when constructing the corresponding adequate test suites.

To determine the adequacy of a test suite, a technique usually involves two components. First, it should statically analyze the set of program elements to be exercised by the test suite. Second, it should dynamically monitor the executions of selected test cases against the program to determine the coverage achieved with respect to the exercised program elements.

It is well known that static analysis to compute the data definition-use associations in a program suffers from severe problems [Santelices and Harrold 2007] when program variables span across multiple procedures, when variables are aliased, or if the program involves concurrent components. Fortunately, the scalability issues in point-to analysis have been significantly alleviated in recent years [Lhoták and Chung 2011; Kastrinis and Smaragdakis 2013]. They help make the static analysis in data-flow testing more practical.

When applying a data-flow adequacy criterion, it generally requires executing test cases against the program to determine the achieved coverage (Frankl and Weyuker, 1988; Hassan and Andrews 2013). Unfortunately, the overhead of runtime monitoring with respect to data-flow adequacy criteria is very high due to the need for profiling data accesses in the course of program executions. For instance, to profile the data accesses from the executions of C/C++ programs via the popular Pin framework, the overhead can be as high as almost a 100-fold slowdown [Luk et al. 2005]. This high runtime overhead problem is further amplified by the fact that data-flow test adequacy criteria are demanding to be satisfied [Weyuker 1990; Hassan and Andrews 2013]. Hence, blindly applying more test cases to a program may not effectively exercise more program elements that are identified to be not covered by previous executions. This issue still prevents these test adequacy criteria from being adopted by the industry to test large-scale applications [Yang et al. 2009]. Significant research should be conducted on data-flow testing to ensure that CPS applications can be applied in practice.

To address these issues, the present work contributes by studying the application of context diversity. As we have described in Section 1, context diversity is a notion that measures the properties of test cases without exercising the program and does not rely on the source code of the program. Test case selection among a set of candidates can be performed by simply using an effective mechanism to guide the process. With respect to this selection problem, our case study shows that using the *CARS-H* strategy can be significantly more effective than not using it.

Static verification approaches have been proposed to ensure the quality of CPS applications. Specification and Description Language (SDL) [Belina and Hogrefe 1989] and Message Sequence Charts (MSC) [Alur and Yannakakis 1999] are two well-known formal techniques to model the interactions among system components of pervasive software. They use formal analysis tools (such as the ObjectGeode[1] tool set) to verify specific objectives (such as deadlocks, livelocks, and the possibility of reaching a particular state during exploration). Roman et al. [1997] modeled a mobile software application in Mobile UNITY and verified the model against the specified properties, particularly the mobility aspects of the software. Murphy et al. [2006]

[1] Available at http://www.verilog.org/.

proposed representing contexts as tuples, and contexts are captured when constraints on contexts are activated. Sama et al. [2010] verified the conformance between the adaptive behavior of a program and a set of proposed patterns, which is concerned with discovering faults in context awareness and adaptation behavior of an application. By dividing the whole adaptive program into nonadaptive functional parts and adaptive ones, Zhang and Cheng [2006] proposed a model-driven approach using Petri Nets for developing adaptive systems. Zhang et al. [2009] also presented a modular A-LTL (an extension of Linear Temporal Logic with Adapt operators) model-checking approach to verifying the adaptation properties of such systems. Kacem et al. [2009] proposed a coordination protocol for distributed adaptation of component-based systems. They then used colored Petri nets to model the key behavioral properties of coordination and conducted CTL (Computational Tree Logic) model checking to assess the correctness of the models and protocols. Xu et al. [2013] automatically inferred domain and environment models to suppress false alarms in state transitions for rule-based context-aware systems. Yang et al. [2013] employed logical time to model the temporal evolution of environment states as a lattice, used a formal language to model the specification of dynamic properties over the traces of environment state evolution, and applied the SurfMaint algorithm to achieve runtime maintenance of the active surface of the lattice. Static verification techniques, however, usually suffer from the scalability issue: they are only suitable for small programs but not large-scale applications [D'Silva et al. 2008]. Different from these verification efforts, our technique does not assume the presence of model-based artifacts and use runtime coverage information and context diversity to facilitate the testing of CPS applications.

The idea of evaluating the quality of a test suite is not new. Harder et al. [2003] proposed adding test cases to a test suite incrementally until the operational abstraction of the test suite is not changed by the next candidate test case. On the other hand, our technique selectively replaces an existing test case of a test suite by a candidate test case if the context diversity of the test suite can be enhanced by such a replacement. Jeffrey and Gupta [2007] employed multiple coverage-based testing criteria to resolve tie cases. A key difference between our strategy and their work is that they tried to retain all redundant test cases with respect to the primary test adequacy criterion as long as they satisfy additional requirements with respect to some complementary criteria. Lin and Huang [2009] proposed to control the sizes of the reduced test suites by picking up only one test case that contributes the most to the complementary criteria from all redundant test cases with respect to the primary test adequacy criterion. Still, the reduced test suite produced by their algorithm may be redundant with respect to the primary criterion. In contrast, the essential idea of our proposed strategy is to replace one test case by another — it does not retain any redundant test case with respect to any test adequacy criterion. More importantly, our technique introduces much less overhead to testers because they can derive context diversity from test inputs without program execution, while all the other techniques discussed in this paragraph need to execute programs to collect runtime white-box information.

Last but not least, the strategy proposed in this article is open to a variety of choices of other test adequacy criteria. For instance, one may integrate our approach with the ideas of Heimdahl and George [2004] to obtain testing items (such as variables, transitions, and conditions/decisions) defined by formal software specifications. One may also combine our approach with the idea of von Ronne [1999] to derive requirements that each testing item needs to be covered multiple times before it is considered sufficiently exercised. In addition, one may also combine our approach with the failure-pursuit sampling strategy proposed by Leon and Podgurski [2003]. That is, after clustering all test cases based on their execution profiles, one test case in each cluster is randomly selected to execute; if it succeeds to find a failure, its k nearest neighbors (which may cross the boundary of clusters) are also selected to execute. Thus, the idea of our approach is general and versatile.

3. FUNDAMENTAL CONCEPTS AND PROPOSED STRATEGIES

In this section, we present the fundamental concepts and notation in our model for CPS applications and testing, as well as our proposed strategies.

3.1 Context-Aware Pervasive Software

A *context variable* v is a characterization of the contexts [Lu et al. 2006, 2008]. We follow Xu et al. [2010] to model a context variable as a tuple $\langle field_1, field_2, ..., field_u\rangle$ such that each $field_w$ (w = 1, 2, ..., u) is an environmental attribute of a CPS application. A *context instance* (denoted by $ins(v)$) is an instantiated context variable such that every field in v is given a value. A *context stream*, denoted by $cstream(v)$, is an input to a CPS application. It is a time series of the form $\langle ins(v)_{t_1}, ins(v)_{t_2}, ..., ins(v)_{t_m}\rangle$, where each $ins(v)_{t_s}$ (for s = 1, 2, ..., m and $t_s < t_{s+1}$) in $cstream(v)$ is a context instance sampled at time t_s.

For example, the smartphone mentioned earlier has a two-dimensional context variable ⟨*location, activity*⟩. When a user presents a report in a meeting room, the context variable is initialized as a context instance ⟨meeting room, present report⟩. As time goes on, the context stream sequence captures a series of activities, beginning with ⟨meeting room, present report⟩, followed by ⟨meeting room, discuss⟩, and finally ⟨home, watch football⟩.

3.2 Context Diversity

Context diversity (denoted by *CD*) [Wang and Chan 2009] measures the number of context changes inherent in a context stream. For any given context stream $cstream(v)$, it computes the total Hamming distance[2] [Hamming 1950; Forney 1966] between all pairs of consecutive context instances, and is defined as:

$$\sum_{i=1}^{L-1} HD(ins(v)_i, ins(v)_{i+1}).$$

Each $HD(ins(v)_i, ins(v)_{i+1})$ is the Hamming distance between the pair of context instances $ins(v)_i$ and $ins(v)_{i+1}$ for i = 1, 2, ..., L–1, where L is the length of the context stream. Consider the context stream example in Section 3.1. For the context stream (⟨meeting room, present report⟩, ⟨meeting room, discuss⟩, ⟨home, watch football⟩), the Hamming distance between ⟨meeting room, present report⟩ and ⟨meeting room, discuss⟩ is 0 + 1 = 1, and that between ⟨meeting room, discuss⟩ and ⟨home, watch football⟩ is 1 + 1 = 2. Hence, the context diversity of the sequence is given by the total Hamming distance of 3.

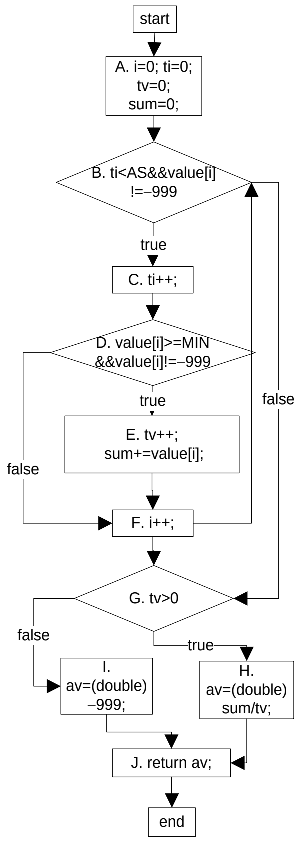
[2] Hamming distance was originally proposed by Hamming [1950] for binary tuples, but was generalized to cover tuples of the form $\langle field_1, field_2, ..., field_u\rangle$ such that the number of possible values in each $field_i$ is finite. See, for example, Forney [1966].

3.3 Data-Flow Testing Criteria

Data-flow testing criteria can be defined in terms of a *Control-Flow Graph* (*CFG*) that models the program structure. A CFG is a directed graph that consists of a set N of nodes and a set $E \subseteq N \times N$ of directed edges between nodes. Each node represents a block of simple statements executed sequentially, and each edge represents execution transfer among nodes. All the nodes in a CFG have both inward and outward edges except the begin node and the end nodes. The begin node has no inward edge since it defines where the execution starts, while an end node has no outward edge since it defines where the execution ends. A *complete* path is a path from the begin node to an end node. For instance, the CFG for *computeAverage*() is shown in Figure 1. It returns the average of all the numbers in the input array within the range [MIN, MAX]. The maximum size of the array is *AS*. The actual array size can be smaller than *AS*, in which case the end of input is represented by –999. For ease of reading, we label each node in Figure 1(b) as *start*, *A*, *B*, *C*, ..., *end*.

```
    double computeAverage(int[] values[], int AS, int
          MIN, int MAX) {
          int i, ti, tv, sum;
          double av;
A.        i = 0; ti = 0; tv = 0; sum = 0;
B.        while (ti < AS && value[i] != –999) {
C.            ti++;
D.            if (value[i] >= MIN && value[i] != –999) {
E.                tv++;
                  sum += value[i];
              }
F.            i++;
          }
G.        if (tv > 0)
H.            av = (double) sum / tv;
          else
I.            av = (double) –999;
J.        return av;
          }
```

(a) Source code



(b) Control-flow graph

Fig. 1. Source code and control-flow graph for *computeAverage*().

Data-flow testing criteria are related to the occurrences of variables within the program. A variable x has a definition occurrence in node n if the value of x is stored in a memory location during the execution of n. A variable x has a use occurrence in node n if the value of x is fetched during the execution of n. A use occurrence of a variable can be further classified as a predicate use (*p-use*) in a condition node (which contains conditional statements such as if-statements or while-statements) or a computation use (*c-use*). For example, the variables *i*, *ti*, *tv*, and *sum* are defined in node *A*, while *ti* has a *p-use* in node *B* and a *c-use* in node *C*.

A path in a CFG is *definition clear* with respect to *x* if none of the nodes in the path (other than the first and the last node) defines *x*. The relation *def_clear*(x, n_i, n_j) denotes a definition-clear path with respect to *x* from n_i to n_j. For this relation, a definition of *x* at node n_i is a *reaching definition* of node n_j and a use of *x* at node n_j is a *reaching use* of node n_i. A *def-use association* (*du-association* for short) is defined as a triple (x, n_i, n_j) such that *x* is used at node n_j, and n_i is a reaching definition of node n_j. The triple (x, n_i, n_j) is covered by a path *p* if *p* is definition clear with respect to *x* and both n_i and n_j are in *p*. For instance, there is a definition-clear path with respect to *values*[] from node *start* to *end* because it does not redefine *values*[]. Hence, we have a du-association (*values*[], *start*, *end*). In contrast, a path with respect to *ti* from node *A* to *C* is not definition clear because *ti* is redefined in node *C*. Furthermore, simple paths and loop-free paths are defined to avoid an infinite number of definition-clear paths in programs with loops. A *simple path* is one in which all nodes, except possibly the first and the last, are distinct. A *loop-free path* is one in which all nodes are distinct.

Frankl and Weyuker [1988] proposed a family of data-flow testing criteria. The criterion *All-Defs* (*AD*) requires that for each definition of the variable *x* in node n_i, there is a complete path that includes a definition-clear path from n_i to n_j such that there is a *c-use* of *x* in n_j or a *p-use* of *x* immediately before n_j. For example, the variable *tv* is defined in node *A* and has a *c-use* in node *E*, and there is a definition-clear path *A–B–C–D–E* from node *A* to *E*. It needs a complete path *start–A–B–C–D–E–F–G–H–J* that includes the definition-clear path *A–B–C–D–E* to satisfy the *AD* criterion. The criterion *all-C-Uses* (*CU*) requires that for each definition of the variable *x* in node n_i, there is a complete path that includes a definition-clear path from n_i to all the nodes n_j such that there is a *c-use* of *x* at n_j or a *p-use* of *x* immediately before n_j. For instance, the variable *ti* is defined in node *A* and has a *c-use* in node *C*, and there is a definition-clear path *A–B–C* from *A* to *C*. We can find a complete path *start–A–B–C–D–E–F–G–H–J* that includes the definition-clear path *A–B–C* to satisfy the *CU* criterion. The criterion *all-P-Uses* (*PU*) is similar to *CU*, except that it exercises all the *p-uses* of the variable in question. For example, the variable *tv* is defined in node *A* and has a *p-use* in node *G*, and *A–B–G* is a definition-clear path. We can find a complete path *start–A–B–G–H–J* that includes the definition-clear path *A–B–G* to satisfy the *PU* criterion. The criterion *all-P-Uses/some-C-Uses* (*PUCU*) is similarly defined. It is identical to the criterion *PU* when the variable has no *c-use*, and reduces the criterion to *some-c-uses* if the variable has no *p-use*, such that for each variable *x*, there are complete paths that include definition-clear paths from the definition of *x* to some nodes that have a *c-use* of *x*. Consider the variable *i* defined in node *A* that has no *p-use* but a *c-use* in statement 10. A complete path *start–A–B–C–D–F–G–H–J* that includes the definition-clear path *A–B–C–D–F* can be used to cover the criterion *PUCU*. Similarly, the criterion *all-C-Uses/some-P-Uses* (*CUPU*) is equivalent to *CU* if a variable has no *p-use*, and reduces to the criterion *some-p-uses* if the variable has no *c-use*. For instance, the variable *value*[] is defined when *computeAverage*() is called and has no *c-use* but a *p-use* in node *D*. A complete path *start–A–B–C–D–E–F–G–H–J* that includes the definition-clear path *A–B–C–D* can be used to cover the criterion *CUPU*. The criterion *All-Uses* (*AU*) produces a set of complete paths due to both the criteria *PU* and *CU*. The criterion *all-DU-paths* is the strictest test adequacy criterion, namely, that for each variable x defined in any node n_i, test cases must traverse complete paths that include (a) all definition-clear simple paths from n_i to all the nodes n_j such that there is a *c-use* of x in n_j and (b) all definition-clear loop-free paths from n_i to all the nodes n_j such that there is a *p-use* of x immediately before n_j.

3.4 Baseline Test Suite Construction Strategy (BS)

We will compare our proposed strategies for enhancing the effectiveness of test suite construction with a baseline strategy based on Frankl and Weyuker [1988]. First, we present the baseline test suite construction strategy in Algorithm 1. It constructs a random test suite with respect to a test adequacy criterion C. It first initializes a test suite S as an empty set. It then randomly selects a test case t from the test pool using the function *randomSelect*(1).[3] We use another function *coverage*: $C \times S \rightarrow [0, 1]$ to return the percentage of coverage items (such as the percentage of program statements for statement coverage) with respect to the criterion C fulfilled by the test suite S. A test suite is said to be *adequate* if the returned value is 100%. If the combined code coverage achieved by $S \cup \{t\}$ against test adequacy criterion C via the function *coverage*(C, $S \cup \{t\}$) is higher than that of S via *coverage*(C, S), the test case t is added to S. The algorithm iterates until it either achieves 100% test coverage or reaches an upper bound M of the number of selection trials. Following the setting of Lu et al. [2008], we set $M = 2000$. For ease of reference, we will refer to Algorithm 1 as *BS*.

ALGORITHM 1: Baseline strategy (*BS*) to construct test suites

Inputs:
M: upper bound of the number of selection trials (a nonnegative integer)
C: test adequacy criterion

Output:
S: *C*-adequate test suite

```
a1: integer trial = 0;                               // no. of selection trials made
a2: S = { };
a3: while (coverage(C, S) < 100% ∧ trial < M) {
a4:     trial ++;
a5:     t = randomSelect(1);                         // randomly select a test case from the test pool
a6:     if (coverage(C, S ∪ {t}) > coverage(C, S))
a7:         S = S ∪ {t};                             // keep the test case in S
a8: }
a9: return S;
```

Suppose, for example, that we would like to construct a test suite S to satisfy the criterion *AU* for *computeAverage*() in Figure 1. At the beginning, *coverage*(*AU*, *S*) = 0 because no test case is included in S (= {}). Suppose *AS*, *MIN*, and *MAX* are global constants that are shared by all test cases and are set to the values of 5, 0, and 100, respectively. Then, a test case $t = [1, 1, –999]$ increases the coverage of S with respect

[3] In general, the function *randomSelect*(k) randomly selects k candidate test cases from the test pool of the benchmark. We assume the existence of a test pool for data flow testing. Interested readers may refer to Edvardsson [1999] for approaches to generate test cases for the test pool.

to *AU* by exercising the complete path *start–A–B–C–D–E–F–G–H–J*, and hence t is included in S (= {t}) based on condition a_6. In contrast, the test case t' = [2, 3, –999] is excluded from S because t' shares the same complete path with t and does not increase *coverage*(*AU*, *S*).

BS not only selects test cases randomly, but also resolves ties randomly. In essence, if both $S \cup \{t\}$ and $S \cup \{t'\}$ achieve the same coverage with respect to a criterion C (i.e., *coverage*(C, $S \cup \{t\}$) = *coverage*(C, $S \cup \{t'\}$)), *BS* makes a random selection between t and t' for inclusion in the updated test suite S.

3.5 Context-Aware Refined Strategies (CARS)

We formulate a family of three strategies, each of which aims at changing the concentration of context diversity in an adequate test suite during the construction of that test suite. We refer to these three strategies as ***CARS-H*** (**C**ontext-**A**ware **R**efined **S**trategy with **H**igh context diversity), ***CARS-L*** (**C**ontext-**A**ware **R**efined **S**trategy with **L**ow context diversity), and ***CARS-E*** (**C**ontext-**A**ware **R**efined **S**trategy with **E**venly-distributed context diversity).

The strategies are presented collectively in Algorithm 2. First, it initializes an empty set S of test cases. It then calls *select*(k, T, *strategy*) to return a test case t. If the coverage achieved by $S \cup \{t\}$ is higher than that of S alone with respect to the given criterion C, the test case t is added to S. The algorithm then iterates until either the coverage achieved by S is 100% or the process has been repeated M times. On the other hand, if the coverage achieved by $S \cup \{t\}$ is *not* higher than that of S alone, Algorithm 2 calls *replace*(C, S, t, *strategy*) to select a test case t' from S, remove t' from S, and add t to S.

The function *select*(k, S, *strategy*) first calls *randomSelect*(k) to construct a candidate test suite T' by randomly selecting k candidate test cases from the test pool and returns one specific test case from T' based on the chosen selection strategy: For *CARS-H* (and *CARS-L*, respectively), it selects an element from T' with the maximum (and minimum, respectively) context diversity. For *CARS-E*, it applies adaptive random testing [Chen and Merkel 2008] that aims at spreading the context diversity values of a test suite evenly and selects an element from T' that maximizes the sum of context diversity differences between this element and every test case in S. Note that if one selects an element from T' randomly, it is simply the *BS* strategy. The function *replace*(C, S, t, *strategy*) is to find a specific test case y from S to be substituted by the candidate test case t identified by *select*(k, S, *strategy*) in order to construct an updated test suite (in lines b_{10} and b_{11} of Algorithm 2), subject to the condition that the coverage of the test suite S before and after this substitution remains unchanged.

The function *replace*(C, S, t, *strategy*) first finds a test case $t1$ that can be substituted by t without affecting the coverage achieved by S (i.e., *coverage*(C, S) = *coverage*(C, $(S \cup \{t\}) \setminus \{t1\}$)) and adds $t1$ to test suite R. It then selects a test case y from R based on the chosen strategy of Algorithm 2: For *CARS-H*, it selects a test case t_L from R with minimum context diversity (i.e., $min(U)$) subject to the condition that $t1$ must exhibit lower context diversity than the given test case t (i.e., $t_L \in R \wedge cd(t) > cd(t_L)$). The selection criterion for *CARS-L* is similar to that of *CARS-H*, except that it selects test cases with higher (instead of lower) context diversity than t from R and then replaces the test case $t1$ that has the maximum (instead of minimum) context diversity with t. For *CARS-E*, it selects all the test cases from R

that have lower distances from the test set $S \cup \{t\}$ than t does and then substitutes the test case $t1$ that has lowest distance from $S \cup \{t\}$ by t.

ALGORITHM 2: Context-aware refined strategy (CARS) to construct test suites

```
CARS(M, C, k)
Inputs:
M: upper bound of the number of selection trials (a nonnegative integer)
C: test adequacy criterion
k: size of candidate test suite
Output: S: C-adequate test suite

b1:  int trial = 0;
        // no. of selection trials
b2:  S = { };
b3:  while (coverage(C, S) < 100% and trial < M) {
b4:      trial ++;
b5:      t = select(k, S, strategy);
b6:      if (coverage(C, S ∪ {t}) > coverage(C, S))
b7:          S = S ∪ {t};
b8:      else {
b9:          t′ = replace(C, S, t, strategy);
b10:         if (t′ is not empty)
b11:             S = (S ∪ {t}) \ {t′};
b12:     }
b13: }
b14: return S;
```

```
select(k, S, strategy)
Inputs:
k: size of candidate test suite
S: test suite under construction
strategy: CARS-H, CARS-L, or CARS-E
Output: x: a test case

c1: $T' = randomSelect(k)$;
c2: if strategy is CARS-H;
c3: $x = t_H$ such that $t_H \in T' \wedge cd(t_H) \geq cd(t)$ for all $t \in T'$;
c4: if strategy is CARS-L
c5: $x = t_L$ such that $t_L \in T' \wedge cd(t_L) \leq cd(t)$ for all $t \in T'$;
c6: if strategy is CARS-E
c7: $x = t_E$ such that $t_E \in T' \wedge D(t_E, S) \geq D(t, S)$ for all $t \in S$, where $D(t, S) = \sum_{s \in S} | CD(t) - CD(s) |$;
c8: return x;
```

```
replace(C, S, t, strategy)
Inputs:
C: test adequacy criterion
S: C-adequate test suite
strategy: CARS-H, CARS-L, or CARS-E,
t: a test case
Output: y: a test case

d1: $R = \{t1 \in S \mid coverage(C, S) = coverage(C, (S \cup \{t\}) \setminus \{t1\}\}$);
d2: if strategy is CARS-H
d3: $y = \min(U)$, where $U = \{t_L \in R \wedge cd(t) > cd(t_L)\}$;
d4: if strategy is CARS-L
d5: $y = \max(U)$, where $U = \{tH \in R \wedge cdt < cdtH\}$;
d6: if strategy is CARS-E;
d7: $y = \min(U)$, where $U = \{t_E \in R \wedge D(t, S \cup \{t\}) > D(t_E, S \cup \{t\})\}$ and $D(t, T) = \sum_{s \in T} | CD(t) - CD(s) |$;
d8: return y;
```

Let us further explain *CARS* using our example in Section 3.5. Suppose we aim to select test cases with high context diversity and hence prefer strategy *CARS-H*. When k is set to a value of 2, we have two candidate test cases [2, 3, –999] and [3, 3, –999] in T and $f1(k, S)$ returns t = [2, 3, –999], which carries the highest context diversity among test cases in T. Suppose S contains one test case t'= [1, 1, –999]. Then, t shares the same complete path with test case t' and does not increase the coverage of S with respect to AU, and hence we enter statements b_9−b_{11} in Algorithm 2. The function $R(AU, \{t'\}, t)$ produces $\{t\}$ because $coverage(AU, \{t'\}) = coverage(AU, \{t\})$ and $CD(t') = 1 < CD(t) = 2$. The function $f2(AU, \{t'\}, t)$ produces t' because t' carries the lowest context diversity among test cases returned by $R(AU, \{t'\}, t)$. (For this particular example, in fact, $R(AU, \{t'\}, t)$ returns only one test case t'.) Finally, we replace t' by t in b_{11} and obtain $S = \{t\}$.

Different from *BS* (which is blind to context diversity), *CARS* uses the context diversity of the test case in two different dimensions. In the data sampling stage, *BS* simply randomly selects a test case from the test pool without reference to any context diversity information (a_5 in *BS*), while *CARS* aims to change the context diversity distribution of the whole test set by favoring different test cases (b_5 in *CARS*()): If the goal is to achieve higher context diversity (c_2 in select()), it favors test cases with the highest context diversity (c_3). It selects test cases with the lowest context diversity (c_5) if it aims to achieve lower context diversity (c_4). It selects test cases with the longest distances from the existing test suite (c_7) if it targets to evenly distribute the context diversity of the test suite. Moreover, when a selected test case cannot improve the overall test coverage achieved by the test suite, *BS* simply drops the test case (a_7 and a_8 in *BS*), while *CARS* uses the context diversity information of the test case to solve the tie case (b_8–b_{12} in *CARS*()): For *CARS-H*, it selects a test case with the lowest context diversity to substitute (d_3). In contrast, *CARS-L* selects a test case with the highest context diversity to replace (d_5). *CARS-E* picks up a test case with the shortest distance from the existing test suite to replace (d_7).

4. CASE STUDY

4.1 Research Questions

We study the following research questions in the case study:

RQ1: How do context changes inherent in individual test cases affect the test effectiveness of coverage-based adequate test suites in ensuring the quality of CPS applications?

RQ2: For test effectiveness, is there any correlation between the context diversity and the white-box measures of the runs?

In general, some adequate test suites can achieve much higher test effectiveness than other adequate test suites with respect to the same test adequacy criterion. It is impossible, of course, to predict the effectiveness of a test suite before execution. On the other hand, testers do not like to put in extra effort in constructing an adequate test suite only to find that the test suite is ineffective. This problem is fundamental and must be addressed. As such, we want to know whether there is any good method that is likely to improve the probability of finding an adequate test suite with higher test effectiveness than a test suite that minimally satisfies the same test adequacy criterion.

RQ1 examines the input value dimension of individual test cases to determine whether this dimension provides a good source of information to steer the selection of adequate test suites with higher test effectiveness (compared with the average case). To the best of our knowledge, our study of RQ1 contributes as the first work to address the problem in the testing of CPS applications.

RQ2 attempts to deepen our understanding of why a black-box metric for CPS applications can affect the fault detection capability of a white-box testing criterion. It is well known that black-box metrics and white-box metrics have the potential to complement each other. Nonetheless, to the best of our knowledge, no empirical result has been reported in the literature (at least for the CPS domain).

4.2 Benchmarks

We used three benchmarks as subjects in our case study: *WalkPath*, *TourApp*, and *CityGuide*, each of which was used as the benchmark in Lu et al. [2008], Wang et al. [2007], and Zhai et al. [2010, 2014], respectively, published in such venues as *International Conference on Software Engineering* and *IEEE Transactions on*

Services Computing. (Note that each of the papers used one subject in its evaluation.) Table I summarizes the descriptive statistics of the three benchmarks.

Table I. Descriptive Statistics of Benchmarks Used in the Case Study

Benchmark	Description	LOC[4]	Middleware	Program Nature	No. of Mutants[5]
WalkPath	Path tracking	803	*LANDMARC*, *Cabot*	Sequential	1,676
TourApp	Tour guide	3,690	*Context Toolkit*	Multithreaded	383
CityGuide	POI recommendation	3,604	*jCOLIBRI*	Sequential	288

The benchmark *WalkPath* is a sequential program. It runs on the *Cabot* middleware [Xu et al. 2010] deployed with a context inconsistency resolution service component. The benchmark also contains a component that implements the classical location-sensing algorithm *LANDMARC* [Ni et al. 2004], which reports a user's location based on RFID data. The program obtains a person's location data by requesting *LANDMARC* to analyze the RFID data, further invokes the context inconsistency resolution services to clean the location data if necessary, and finally reports the person's movement and data reliability based on the (cleaned) location data. Therefore, the only context variable used in *WalkPath* is RFID location data.

The multithreaded benchmark *TourApp* is the largest application shipped with the *Context Toolkit* middleware [Salber et al. 1999]. *Context Toolkit* consists of a number of widgets (such as communication widgets that implement HTTP or TCP protocols to transfer contextual data, assembling widgets that collect related contexts into a group, and subscribe-and-callback widgets that allow the application tier component of a CPS application to register interested context data and specify how to react to changes in the subscribed context data). *TourApp* informs visitors who attend a conference about demos of interest based on the visitors' preferences and location data. In the current empirical setting, *TourApp* uses the visitors' activity information such as ⟨login, enter room, view demo, logout⟩ as the only context variable.

The benchmark *CityGuide* is a sequential program. It runs on the case-based-reasoning middleware *jCOLIBRI* [Diaz-Agudo et al. 2007]. *jCOLIBRI* retrieves the cases from a relational database and reasons about the provided contextual data (which include user preferences such as payment methods, food styles, and room types, as well as GPS location data in terms of latitudes and longitudes) to recommend the best points of interest such as hotels and restaurants based on similarity functions of the contextual data and user decision history. It then saves the users' confirmed decisions in the database. *CityGuide* captures multiple context variables such as the users' preferences (e.g., payment methods, food styles, and room types) as well as GPS location data in terms of latitudes and longitudes. We assign the same weight to each context variable so that they are treated identically when calculating context diversity.

The sizes of *WalkPath*, *TourApp*, and *CityGuide* are 803, 3,690, and 3,604 lines of code, respectively.

[4] Measured by *JavaNCSS*, available at http://javancss.codehaus.org/.
[5] Generated by *MuClipse*, available at http://muclipse.sourceforge.net/.

4.3 Faulty Versions

In mutation analysis [Andrews et al. 2006; Budd et al. 1980], a *program mutant* refers to a variation of a program under test by a small syntactic change. It mimics a simple fault in the program. Previous research such as Andrews et al. [2006] has shown that more complex and real faults in the same program are strongly coupled with these mutants, and test suites that kill these mutants are highly effective to expose real and complex faults in the same program.

The process of mutation analysis involves the execution of test cases against a set of program mutants. When executing a test case against the program under test and executing the same test case against a program mutant produce a difference in program output, the mutant is said to be *killed*.

We refer to the proportion of a test suite that can kill a mutant as the *failure rate* of the mutant with respect to the test suite. We measure this rate as a proxy of the probability that the mutant can be killed by at least one test case in the test suite.

We adopt mutation analysis to evaluate the techniques. We particularly note that, to apply our techniques in practice, there is no need to generate any mutants to test CPS applications.

Specifically, we generated mutants for each of our benchmarks and deemed them as *faulty versions* of the corresponding benchmarks. Initially, we planned to use all the mutants of each benchmark. After running the first 10 mutants of *TourApp*, however, we found it impractical because the test run of each mutant took much time to complete. As such, we refined the methodology for *TourApp* by applying the procedure recommended by Andrews et al. [2006]. Thus, we selected every fifth mutant of *TourApp*. We retained all the mutants whose failure rates fell within the range (0.06, 1.00). Our approach to excluding the results of faulty versions follows the work of Lu et al. [2008]. In total, 475, 244, and 240 mutants were retained for *WalkPath*, *TourApp*, and *CityGuide*, respectively, to be used as faulty versions of the benchmarks for data analysis. The mean failure rates of the retained mutants for *WalkPath*, *TourApp*, and *CityGuide* were 0.311, 0.369, and 0.150, respectively.

4.4 Test Adequacy Criteria

We used six data-flow testing criteria introduced in Section 3.3 in our case study, namely, *AD*, *CU*, *PU*, *PUCU*, *CUPU*, and *AU*. We did not investigate the *all-DU-paths* criterion because the complexity of du-paths was not polynomial with respect to the number of conditional statements encountered, which was intractable. To circumscribe the cost of static analysis for data-flow testing criteria discussed in Section 2, we used a runtime monitoring technique to collect the du-associations for any given criterion: We used the open-source tool *Gretel* [6] to instrument the program under test and monitored the execution path of each test case. Then, following Misurda et al. [2005], we figured out the last definition of each usage of the same memory location in each execution trace to analyze the du-associations of every test case. Different from static analysis techniques, the test requirements inferred by such dynamic analysis were all feasible in the sense that they were all reachable by at least one test case. State-of-the-art test adequacy criteria have been proposed by Lu et al. [2008] to test context-aware software with context inconsistency resolution services. They are applicable only to *WalkPath* but not *TourApp* and *CityGuide*

[6] Available at http://sourceforge.net/projects/gretel/.

because the latter two benchmarks do not have such services defined to clean noisy contexts. To maintain consistency, therefore, we do not study the criteria proposed by Lu et al. in our case study.

4.5 Preparation of Test Cases and Test Suites

We reused an existing test pool for *WalkPath*, which contained 20,000 distinct test cases, each consisting of real-world data captured via RFID readers and used in integration testing experiments [Lu et al. 2006, 2008]. For *TourApp* and *CityGuide*, we developed a random input generation tool to generate 5,000 test cases such that the benchmarks neither raised any unhandled exception nor resulted in infinite loops (and, of course, each benchmark outputs a result in each case). Based on the practical guidelines in Arcuri and Briand [2011], 5,000 test cases are sufficient in terms of randomness to obtain a statistical conclusion.

When constructing adequate test suites for each benchmark, the upper bound M of the number of selection trials for *BS*, *CARS-H*, *CARS-L*, and *CARS-E* was set to 2,000, which was the same as that in Lu et al. [2008]. Similar to the experiment in Lu et al., we also configured our tool to use the random selection method as the test case generator *generate*(1) in Algorithm 1 and *generate*(k) in Algorithm 2.

To conduct a controlled experiment to explore how context diversity affects test effectiveness, we need to ensure that the test suites for comparison constructed by different strategies (namely, *BS*, *CARS-H*, *CARS-L*, and *CARS-E*) achieve the same coverage and are of the same size. We systematically varied the sizes of candidate test suites for k = 1, 2, 4, 8, 16, 32, 64, and 128. Our tool constructed 1,000 adequate test suites for each combination of test adequacy criterion, test suite construction strategy, and size of the candidate test suite. As a result, 144,000 test suites were constructed for each benchmark. For every criterion and every benchmark, we collected the set of test suite sizes such that each can be attained by all adequate test suites with respect to the given criterion regardless of the test suite construction strategy (*BS*, *CARS-H*, *CARS-L*, or *CARS-E*) and the value of k (1, 2, 4, 8, 16, 32, 64, or 128). We then picked up the largest among the set of test suite sizes as the *baseline test suite size* for the criterion. Thus, the baseline test suite size for a specific test adequacy criterion can be shared by all its adequate test suites irrespective of test suite construction strategies and the setting of the sizes of candidate test suites. We show the baseline test suite sizes for each criterion in Table II.

Finally, for each benchmark, we constructed 100 test suites for each combination of test adequacy criterion, context-aware refined strategy, and size of candidate test suite. As such, we obtained 14,400 (= 100 × 6 × 3 × 8) adequate test suites for each benchmark for the purpose of test effectiveness comparison. It took 1.5 months to construct test suites for all the four strategies (*CARS-H*, *CARS-L*, *CARS-E*, *BS*), eight values of k (1, 2, 4, 8, 16, 32, 64, 128), six data-flow testing criteria (*PU*, *PUCU*, *CU*, *CUPU*, *AD*, *AU*), and three subjects (*WalkPath*, *CityGuide*, *TourApp*). In the best setting such that *CARS* outperformed *BS* the most (corresponding to the *CARS-H* strategy, k = 64, and the *PUCU* testing criterion), it took about 1, 2, and 4 hours to construct 100 adequate test suites for *WalkPath*, *CityGuide*, and *TourApp*, respectively. The differences among subjects were due to different modes of program executions. For example, *TourApp* employs the client/server model to implement the business logic. A typical communication between clients and servers is as follows: the clients send context instances to the server via the network, and then the server analyzes the context instances and returns the analysis results to the clients. Since network communications usually take time, the execution of *TourApp* was the most time-consuming. *CityGuide* accessed a database and hence its execution was faster than *TourApp*, while *WalkPath* did not access any database or network and its execution was the fastest.

Compared with the experiment by Lu et al. [2008], which uses one benchmark and four criteria with 30 faulty versions, our experiment is significantly larger by two orders of magnitude.

Table II. Baseline Test Suite Sizes for each Criterion

	AD	*CU*	*PU*	*PUCU*	*CUPU*	*AU*
WalkPath	10	15	5	10	16	18
CityGuide	9	13	5	5	13	14
TourApp	12	18	12	16	19	24

4.6 Construction of Interacting Finite State Machine

We used a Finite State Machine (FSM) to model the interactions between a CPS application and its computing environment for the investigation of RQ2. For *WalkPath*, whenever a context inconsistency resolution service [Lu et al. 2008] was triggered to clean data, we modeled the process as a transition from the current location to the resolved location and labeled the transition with the input location. For *TourApp*, whenever a new user activity was captured, we modeled the process as a transition from the current user activity to the new one, and labeled the transition with the user activity. For *CityGuide*, whenever a returned location was confirmed by the user, we modeled the process as a transition from the current location to the confirmed location, and labeled the transition with the outputted location. For each benchmark, we constructed an interacting FSM from the corresponding test pool so that the FSM can represent all the behavior that could be exercised by the test pool against the benchmark. Table III shows the sizes of the interacting FSMs constructed for the three benchmarks.

Table III. Sizes of Interacting FSM for Benchmarks

Benchmark	No. of States	No. of Transitions
WalkPath	30	133
CityGuide	8	50
TourApp	4	16

4.7 Hardware Platform

We conducted the case study on 16 high-performance cluster nodes from the HKU Gideon-II Cluster. Each cluster node was equipped with 16 x 2.53GHz quad-core Xeon processors with 16GB physical memory. The operation system for all the cluster nodes was Linux version 2.6.9-82.ELsmp 64-bit. This cluster machine (without any GPU usage) achieved the performance of 3.45 TFLOPS on the Linpack benchmark with a peak performance of 5.181 TFLOPS. Each benchmark was written in Java and executed under JRE version 1.6.0_23-b05 with the use of Java HotSpot (TM) Server VM (build 19.0-b09, mixed mode).

4.8 Experimental Procedure

We first followed the procedure presented in Section 4.5 to prepare the test pool and test suites for each benchmark. For each test case, we collected its context diversity and the execution path length. Furthermore, we computed the test effectiveness of each test adequacy criterion to analyze how it is affected by context diversity. Then, we used *MuClipse* version 1.3[7] to generate a set of mutants for the benchmarks and ruled out any syntactically equivalent mutants.

5. DATA ANALYSIS

This section presents the experimental results that tackle the research questions raised in Section 4.1. More specifically, Section 5.1 investigates how our strategies affect the context diversity of test suites, which validates the design goals of the strategies. Section 5.2 addresses RQ1 that studies how context diversity affects the effectiveness of test suites, and Sections 5.3 and 5.4 address RQ2 by correlating context diversity with white-box testing criteria, namely, execution path lengths and context-aware system interactions.

5.1 Effects of Different Strategies on Context Diversity

We first present the context diversity of test suites constructed by the *BS* strategy in Table IV. The data shows that stronger test adequacy criteria (in terms of the subsumption relationship proposed by Frankl and Weyuker [1988]) do not necessarily exhibit higher context diversity than weaker test adequacy criteria. For example, criterion *PUCU* is stronger than criterion *PU*, but for *WalkPath*, test suites constructed by *PUCU* carry lower context diversity than test suites constructed by *PU*. Similar observations can be found for *CityGuide* and *TourApp*. For instance, the context diversity of test suites constructed by *PUCU* is the same as that of test suites constructed by *PU* for *CityGuide*, whereas test suites constructed by a stronger criterion *AU* carry lower context diversity than test suites constructed by a weaker criterion *AD* for *TourApp*. Furthermore, we find that test suites constructed by different test adequacy criteria share very similar context diversity values. For example, the largest difference in context diversity for test suites constructed by different criteria is less than 0.3 for *WalkPath*, the largest difference never exceeds 2.1 for *CityGuide*, and the largest difference is 0.9 for *TourApp*. These observations suggest that test adequacy criteria may be not a significant factor that affects the context diversity of their adequate test suites.

Table IV. Context Diversity of Test Suites Constructed by *BS*

Test Adequacy Criterion	*WalkPath*	*CityGuide*	*TourApp*
PU (*all-P-Uses*)	12.1	13.6	15.5
PUCU (*all-P-Uses/some-C-Uses*)	11.8	14.3	16.1
CU (*all-C-Uses*)	11.9	12.2	15.7
CUPU (*all-C-Uses/some-P-Uses*)	12.1	12.2	15.9
AD (*All-Defs*)	11.8	12.7	16.4
AU (*All-Uses*)	12.1	12.9	15.6

Figure 2 summarizes to what extent context diversity can be incorporated into adequate test suites through different strategies with respect to the baseline strategy *BS*.[8] Starting from top left and then clockwise, these nine subfigures show the differences in context diversity of *C*-adequate test suites under different sizes (k = 1, ..., 128) of the candidate sets compared with the context diversity using the *BS*

[7] Available at http://muclipse.sourceforge.net/.
[8] Interested readers may refer to Figures A1−A3 in the Online Appendix for the data and analysis that produce the curves in the figure.

strategy, where criteria *C* are *PU*, *PUCU*, *CU*, *AU*, *AD*, and *CUPU*, respectively. For instance, in the top left subfigure, when the candidate size is 1, the *CARS-H* for *TourApp* is higher in context diversity than *BS* by 6.185.

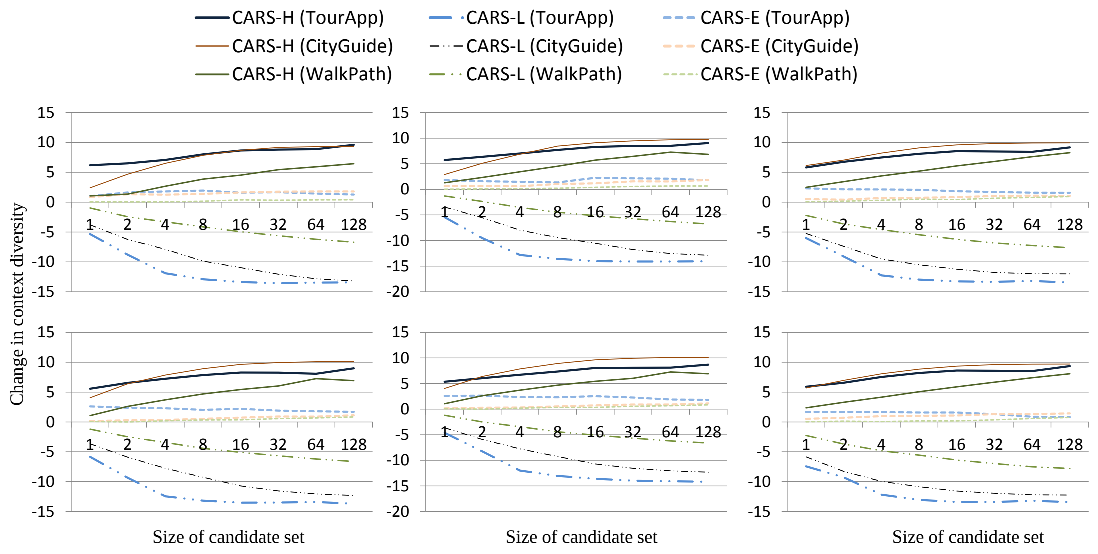


Fig. 2. Differences in context diversity between *CARS* and *BS*.
From top left and then clockwise are the results of *PU*, *PUCU*, *CU*, *AU*, *AD*, and *CUPU*, respectively.

In each subfigure, we observe that all the solid lines are above the *x*-axis, and generally show upward trends as the *x*-value increases. It indicates that *CARS-H* consistently achieves higher context diversity than *BS*, and the difference is increasingly more noticeable with larger values of *k*. This observation follows the design of *CARS-H* because every test case substitution made by *CARS-H* improves on the context diversity of the adequate test suite being constructed, and a larger candidate test suite should offer a high probability of a successful substitution. The dashed-and-dotted lines are always positioned below the *x*-axis, and generally show downward trends as the *x*-value increases. It shows that the *CARS-L* strategy always results in worse context diversity. Moreover, we find that the difference in context diversity resulting from the use of *CARS-H* and *CARS-L* on each test adequacy criterion can be as large as 50%−90% when the size of a candidate test suite is relatively large (such as k = 32 or higher). The finding shows that it is feasible to significantly modify the context diversity of an adequate test suite, which is encouraging.

In Figure 2, *CARS-E* also leads to higher context diversity than *BS*, but the magnitude is generally much less noticeable than *CARS-H* and *CARS-L*. The finding indicates that the idea of evenly spreading test cases in the context diversity dimension exhibits less effectiveness than *CARS-H* or *CARS-L* in the case study.

Table V. No. of Cases that Reject the Null Hypothesis for Context Diversity Comparison for Various *CARS* Strategies and Benchmarks

Benchmark	*CARS-H*	*CARS-L*	*CARS-E*
WalkPath	44	45	0
CityGuide	46	48	0

TourApp	48	48	0

To investigate whether the difference between various test suite construction strategies and the baseline strategy is significant, we conduct hypothesis testing for each test suite construction strategy with the null hypothesis "test suites constructed by a specific context-aware refined strategy share the same context diversity with those constructed by the baseline strategy *BS*" at a significance level of 5%. We present the summarized results in Table V. Interested readers may find the detailed data from Table A2 in the Appendix.

For each benchmark and for each *CARS* strategy, we evaluate 48 different cases corresponding to all combinations of the six different criteria (namely, *PU*, *PUCU*, *CU*, *CUPU*, *AD*, *AU*) and eight different sizes of candidate sets (k = 1, 2, 4, 8, 16, 32, 64, 128). Each cell in Table IV shows the number of cases that successfully rejects the null hypothesis. For example, for *WalkPath* using *CARS-H*, there are 44 out of 48 cases successfully rejecting the null hypothesis. Table V shows that, regardless of benchmarks and test adequacy criteria, both *CARS-H* and *CARS-L* are more likely to reject the null hypothesis. For example, *CARS-H* rejects the null hypothesis for 44, 46, and 48 out of 48 cases on *WalkPath*, *CityGuide*, and *TourApp*, respectively. *CARS-L* rejects the null hypotheses for 45, 48, and 48 out of 48 cases on *WalkPath*, *CityGuide*, and *TourApp*, respectively. In contrast, *CARS-E* consistently fails to reject the null hypothesis for candidate test suites of all sizes. This result validates that both *CARS-H* and *CARS-L* have a significant effect on the context diversity of the adequate test suites thus constructed, and the effect of *CARS-E* on the context diversity of test suites is not statistically significant.

To sum up, stronger testing criteria do not necessarily lead to higher context diversity, which motivates us to develop new strategies (such as *CARS* strategies presented in this article) to alter the mean context diversity of a test suite during the construction of the test suite. Moreover, *CARS-H* (and *CARS-L*, respectively) tends to result in higher (and lower, respectively) context diversity with the increase of the value of k. The observation shows that our strategies successfully change the context diversity distribution of test suites. In Sections 5.2 to 5.4, we will further use these test suites for the data analysis to answer RQ1 and RQ2.

5.2 Effects of Different Strategies on Overall Test Effectiveness

We present in Table VI the mean fault detection rate of each criterion using the baseline strategy *BS*. In the table, for each of the three benchmarks, *AU* consistently outperforms *PUCU* and *CUPU*, *PUCU* outperforms *PU*, and *CUPU* outperforms *CU* in terms of attaining higher mean fault detection rates averaged over all mutants. This finding is in line with the popular understanding on test adequacy criteria that stronger criteria are generally more effective than weaker ones in exposing faults.

Table VI. Fault Detection Rates of Test Suites Constructed by *BS*

Test Adequacy Criterion	*WalkPath*	*CityGuide*	*TourApp*
PU (*all-P-Uses*)	0.218	0.163	0.651
PUCU (*all-P-Uses/some-C-Uses*)	0.414	0.247	0.752
CU (*all-C-Uses*)	0.622	0.264	0.812
CUPU (*all-C-Uses/some-P-Uses*)	0.218	0.163	0.651
AD (*All-Defs*)	0.414	0.247	0.752
AU (*All-Uses*)	0.622	0.264	0.812

Figure 3 summarizes the changes in test effectiveness between the baseline strategy *BS* and a context-aware refined strategy (*CARS-H*, *CARS-L*, and *CARS-E*, respectively).[9] Each subfigure can be interpreted in the same way as a subfigure in Figure 2, except that the *y*-axis now stands for the change in fault detection rate instead of the change in context diversity.

We observe from Figure 3 that all the solid lines in each subfigure are above the *x*-axis. This indicates that *CARS-H* consistently improves the testing effectiveness in terms of the mean fault detection rates regardless of benchmarks, test adequacy criteria, and sizes of candidate test suites. Moreover, the effect of the size of the candidate test suite seems to be saturated at $k = 64$ in the experiment. In particular, when $k = 64$, the improvements in test effectiveness for all criteria induced by *CARS-H* over *BS* are 10.6%–21.9% for *WalkPath*, 12.3%–22.1% for *TourApp*, and 12.6%–14.5% for *CityGuide*, respectively. In summary, irrespective of benchmarks and criteria, when $k = 64$, *CARS-H* can bring about an improvement of 10.6%–22.1% in test effectiveness to the baseline strategy in terms of mean fault detection rates. Considering that data-flow testing is by itself a highly effective technique, the result shows that the improvement made by *CARS-H* is significant.

Moreover, irrespective of benchmarks and test adequacy criteria, the lines for the *CARS-L* strategy are always below the *x*-axis, indicating that this strategy consistently performs less effectively than the *BS* strategy in terms of mean fault detection rate.

Combined with the result presented in Section 5.1, we find that changing the context diversity of an adequate test suite via *CARS-H* and *CARS-L* *does* correlate with the change in test effectiveness. In contrast, *CARS-E* has less impact on test effectiveness compared with *CARS-H* and *CARS-L*.

Similar to Table V, we also summarize the hypothesis testing results in Table VII. Detailed data can be found in Tables A7−A9 in the Appendix. The null hypothesis is "test suites constructed by a specific context-aware refined strategy share the same fault detection rate with those constructed by the baseline strategy *BS*" and the significance level is 5%.

[9] Interested readers may refer to Figures A4−A6 in the Online Appendix for the detailed data analysis.

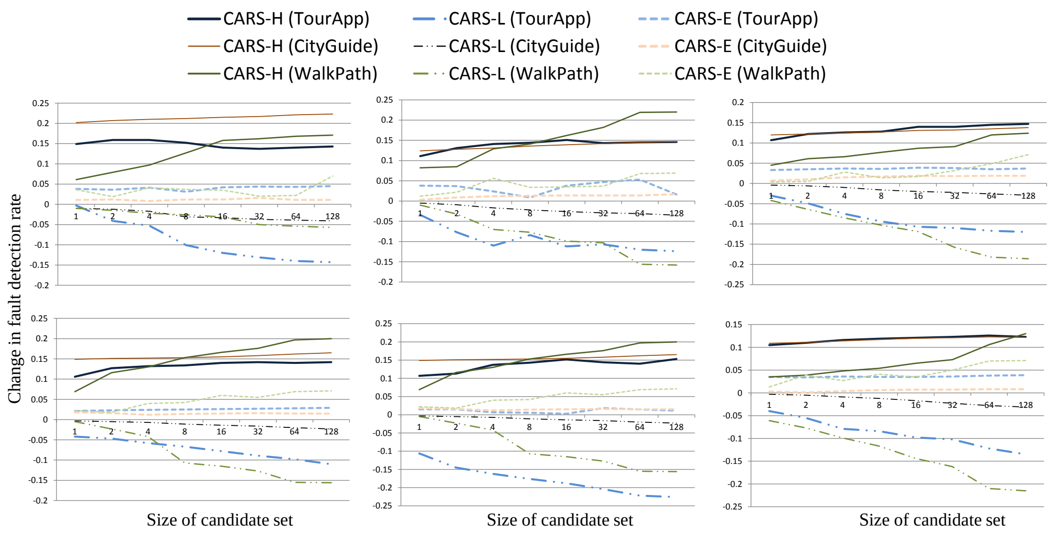


Fig. 3. Differences in fault detection rates between *CARS* and *BS*.
From top left and then clockwise are the results of *PU*, *PUCU*, *CU*, *AU*, *AD*, and *CUPU*, respectively.

Table VII. No. of Cases that Reject the Null Hypothesis for Testing Effectiveness Comparison for Various *CARS* Strategies and Benchmarks

Benchmark	*CARS-H*	*CARS-L*	*CARS-E*
WalkPath	30	28	0
CityGuide	48	0	0
TourApp	48	38	0

The hypothesis testing results confirm that *CARS-H* can outperform the *BS* strategy significantly in terms of the fault detection rates achieved by test suites with a specific candidate suite size and coverage. For example, *CARS-H* rejects the null hypothesis for 30, 48, and 48 out of 48 cases on *WalkPath*, *CityGuide*, and *TourApp*, respectively. In contrast, *CARS-L* deteriorates the effectiveness of test suites significantly. For example, *CARS-L* rejects the null hypotheses for 28 and 38 out of 48 cases on *WalkPath* and *TourApp*, respectively. We also find that quite a number of outliers exist for *CityGuide*: *CARS-L* fails to reject the null hypothesis for all these cases. As shown in Table VI, owing to the low fault detection rates achieved by test suites constructed by the baseline strategy *BS*, it would be hard for *CARS-L* to deteriorate the test effectiveness further. In contrast, *CARS-E* consistently fails to reject the null hypothesis for candidate test suites of all sizes. This result validates that both *CARS-H* and *CARS-L* can have significant impacts on testing effectiveness of the adequate test suites constructed. The impact of *CARS-E* on the testing effectiveness of test suites is not statistically significant compared with the test suites constructed by the *BS*.

In this section, we have shown that across all benchmarks and all values of *k*, the *CARS-H* (and *CARS-L*, respectively) strategy can significantly improve (and deteriorate, respectively) the effectiveness of the evaluated data-flow testing criteria. This result answers RQ1 directly: context diversity of test cases can significantly correlate with the test effectiveness of data-flow adequate test suites for CPS applications. This finding leads to two suggestions for test engineers when they test CPS software: (a) better test effectiveness can be achieved by favoring test cases with higher context diversity and (b) context diversity helps resolve tie cases in which multiple test cases have the same contribution to test coverage. Note that it does not need any code change of CPS software to apply *CARS* because context diversity can be measured without knowledge of the source code. Moreover, we are aware of context diversity before the execution of test cases, so our *CARS* strategies do not need runtime profiling to access the current context diversity during program execution.

5.3 Effects of Different Strategies on the Normalized Execution Path Length

Figure 4 presents the normalized execution path lengths of test suites constructed by different strategies.[10] Each subfigure can be interpreted in the same way as a subfigure in Figure 2, except that the *y*-axis is now the change in normalized execution path length instead of the change in context diversity.

Table VIII. Mean Normalized Execution Path Length of Test Suites Constructed by *BS*

Test Adequacy Criteria	*WalkPath*	*CityGuide*	*TourApp*
PU (*all-P-Uses*)	0.464	0.361	0.681
PUCU(*all-P-Uses/some-C-Uses*)	0.473	0.369	0.688
CU(*all-C-Uses*)	0.481	0.374	0.696
CUPU(*all-C-Uses/some-P-Uses*)	0.484	0.377	0.698
AD(*All-Defs*)	0.465	0.365	0.682
AU(*All-Uses*)	0.492	0.389	0.709

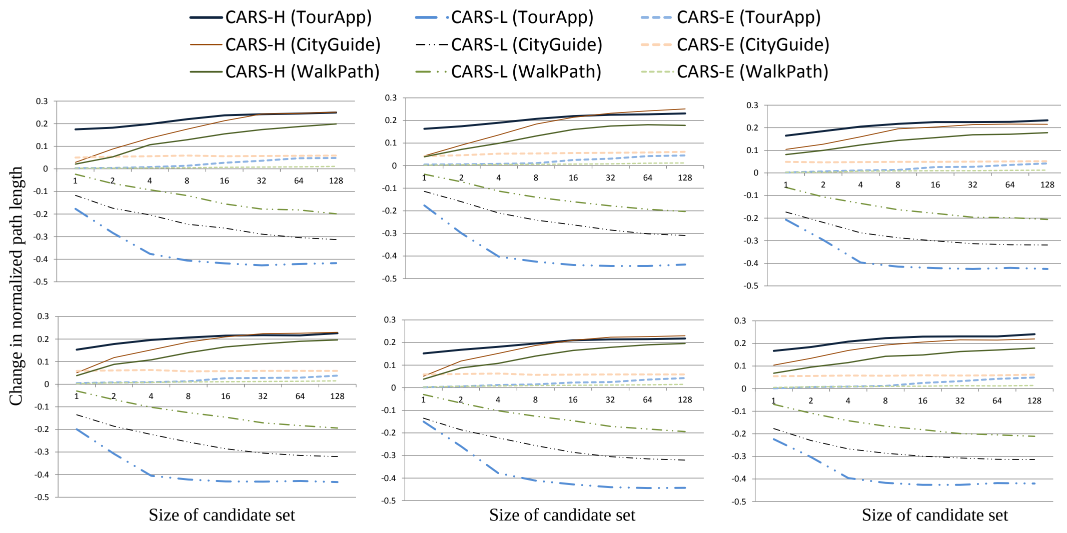


Fig. 4. Differences in normalized execution path lengths between *CARS* and *BS*.
From top left and then clockwise are the results of *PU*, *PUCU*, *CU*, *AU*, *AD*, and *CUPU*, respectively

We first present the mean normalized execution path lengths of test suites constructed by *BS* in Table VIII. We define a normalized execution path length as the ratio of the length of that path to the length of the longest execution path exercised

[10] Interested readers may refer to Figures A7–A9 in the Online Appendix for the data and analysis that produce the curves in the figure.

by the test cases in the test pool. We find that for all the benchmarks, *AU* always leads the programs to execute the longest paths among all test adequacy criteria, followed by *CUPU*, *CU*, *PUCU*, *AD*, and finally *PU*. This order is consistent with the test effectiveness performance of the test adequacy criteria reported in Table VI, which further suggests that the normalized execution path length contributes to the effectiveness of test suites.

We find that for every subfigure in Figure 4, the solid lines are consistently above the *x*-axis. This shows that for each size of candidate test suites, *CARS-H* consistently exercises more statements than *BS*. In contrast, the lines for *CARS-L* are always positioned below the *x*-axis. It shows that these test cases execute shorter paths than *BS* on average. Moreover, *CARS-E* leads to longer execution paths than BS, but only has a much lighter effect than *CARS-H* in forcing test cases to traverse longer paths.

Table IX. Number of Cases that Reject the Null Hypothesis for Comparing Normalized Execution Path Lengths for Various *CARS* Strategies and Benchmarks

Benchmark	*CARS-H*	*CARS-L*	*CARS-E*
WalkPath	44	39	0
CityGuide	44	48	0
TourApp	48	48	0

Similar to the previous two subsections, we also conduct hypothesis testing to confirm our observation. The null hypothesis is "test suites constructed by a specific context-aware refined strategy share the same normalized execution path lengths with those constructed by the baseline strategy *BS*" and the significance level is 5%.

The results of the hypothesis testing are shown in Table IX. *CARS-H* rejects the null hypothesis for 44, 44, and 48 out of 48 cases on *WalkPath*, *CityGuide*, and *TourApp*, respectively. *CARS-L* rejects the null hypothesis for 39, 48, and 48 out of 48 cases on these three benchmarks, respectively. However, *CARS-E* fails to reject any null hypothesis in all cases. We find that the effect of *CARS-H* is more consistent than that of *CARS-L* (albeit in modifying the normalized execution path lengths in the opposite directions), and *CARS-E* appears to be neutral.

In summary, to answer RQ2, we find that test suites having higher context diversity can lead to longer execution paths. Since various test suites with different context diversity may be constructed to achieve the same code coverage, code coverage appears not to be an explanation for the better effectiveness observed in the case study. In contrast, as shown in our previous work [Wang et al. 2010], the execution times and execution sequence of context-related statements can correlate with test effectiveness, and longer execution paths may lead to more diverse execution times. Our results thus indicate that in the domain of CPS applications, apart from test adequacy criteria, context changes inherent in test suites can be a promising research dimension to harvest so as to improve the effectiveness of the test suites constructed.

5.4 Effects of Different Strategies on System Interactions

In this section, we report to what extent the context diversity of a test case correlates with the state coverage and transition coverage achieved by the test case against the corresponding interacting FSM.

Different programs have different ranges of context diversity. To facilitate comparisons across the three benchmarks, we have normalized the context diversity of each test case by dividing the actual context diversity by the maximum *e* for the respective benchmark. For brevity, we will refer to such kind of context diversity as normalized context diversity.

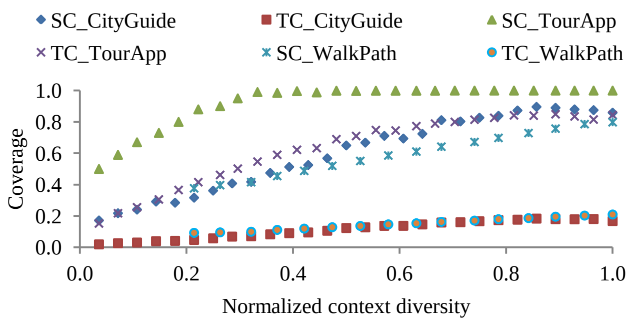


Fig. 5. State coverage (SC) and transition coverage (TC) vs. normalized context diversity

We report in Figure 5 how the normalized context diversity correlates with the state and transition coverage of interacting FSMs. In Figure 5, the state coverage increases from 0.46 to 0.80 (by a factor of 1.7) when the normalized context diversity increases from 0.37 to 1.00 (by a factor of 2.7) for *WalkPath*. For *CityGuide*, the state coverage increases from 0.48 to 1.00 (a factor of 2.1) when the normalized context diversity increases from 0.36 to 1.00 (a factor of 2.8). For *TourApp*, however, the state coverage can only increase from 0.99 to 1.00 (a factor of 1.01) when the normalized context diversity increases from 0.33 to 1.00 (a factor of 3.0).

In addition, we conduct Pearson's correlation test [Pearson 1920] to determine the strength of the correlation if a linear regression trend indeed exists between the two variables. Although there is no golden criterion to interpret the results of Pearson's correlation tests in software engineering research, we use the following definition: If the absolute value of Pearson's Correlation Coefficient (PCC) is greater than 0.8, the correlation is regarded as *strong*. If the absolute value is more than 0.1 but less than 0.5, the correlation is considered *mild*. If the absolute value is at most 0.1, there is *no* correlation. Otherwise, the correlation is said to be *moderate*. A similar interpretation is also used in the experiment in Binkley and Harman [2004]. The PCC result for the correlation between mutation scores and statement coverage is shown in Table X. The results in the table consistently show that, regardless of the benchmark, PCC values estimated by each metric are higher than 0.7. This suggests that context diversity of a test case has a moderate to strong correlation with both state coverage and transition coverage.

Table X Pearson's Correlation Coefficient between context diversity and state coverage and transition coverage of interacting FSMs

Coverage	*WalkPath*	*CityGuide*	*TourApp*
State	0.998	0.981	0.745
Transition	0.998	0.980	0.945

In summary, to answer *RQ*2, we find that there exists a moderate to strong correlation between context diversity and state and transition coverage of interacting FSMs. It suggests that higher context diversity tends to increase the interactions between CPS applications and the computing environment.

5.5 Threats to Validity

Threats to construct validity. Construct validity relates to whether our defined metrics really measure the properties we intend to capture. We used fault detection rate to measure the effectiveness of a test suite. The use of other metrics such as the time needed to generate an adequate test suite may produce different comparison results. To minimize potential threats from inherent stochastic fault detection behavior of individual test suites, we constructed a sufficiently large number of adequate test suites per test adequacy criterion to compute the corresponding fault detection rates.

Another concern is that we use mutation faults rather than real-life faults in our experiment. The "competent programmer hypothesis" in mutation analysis is supported by both empirical and theoretical results [Budd et al. 1980]. Andrews et al. [2006] have further validated that test results of coverage-based testing criteria with respect to mutation faults can be generalized to those with respect to real-life faults.

Threats to internal validity. Internal validity is concerned with whether there is any bias in the experimental design that can affect the causal relationship under study. We have implemented a tool to construct test suites and computed the context diversity and coverage information of test suites, as well as the failure rates of mutants. The correctness of this tool would determine whether there is any potential implementation bias in our experiment. We have tested the tool with small programs and spot-checked the results for larger programs to ensure that it works as expected. The test pool used in the experiment can also introduce a bias. Different test case generation techniques may introduce different biases in the results due to differences in the levels of code coverage. Because we focus on assessing the impact of context diversity on the fault detection capability of test adequacy criteria, we employ the random test case selection method used in the experiment of Lu et al. [2008] as the test input generators *generate*(1) in Algorithm 1 and *generate*(k) in Algorithm 2. We only retain those generated mutants with failure rates higher than 0.06 as candidate faulty versions of the benchmarks in the case study. The use of mutants with different failure rates may produce different results. To ensure that the experiment ends within a manageable time, we follow the sampling strategy in Andrews et al. [2006] to include every fifth mutant of *TourApp* in the experimentation. The inclusion of more mutants would significantly lengthen the experimentation. To strike a balance between the effort and the representativeness of the experiment, we settle for the current strategy.

We have used random test case selection as the baseline strategy for comparison. The use of more advanced strategies as the baseline would produce different improvement results. However, according to Harman and McMinn [2010], "sophisticated search techniques such as Evolutionary Testing [most commonly implemented as genetic algorithms] can often be outperformed by far simpler search techniques." In their paper, the authors proposed hill climbing as a "far simpler search technique" and found that "Where test data generation scenarios do not have a Royal Road property, Hill Climbing performs far better than Evolutionary Testing." The *CARS* algorithms in our article are similar to hill climbing, except that *CARS* terminates after a number of trials instead of one trial.

Threats to external validity. External validity refers to the extent to which we can generalize our empirical results to other benchmarks. In our case study, we use the 40 Java mutation operators proposed in Ma et al. [2006] to generate candidate mutation faults for experimentation. Other mutation operators for different programming languages may produce different results. We have included three benchmarks in our case study. They do not represent all types of CPS applications. Although the use of more benchmarks would certainly increase the power of data analysis, the current experiment has taken more than three months to run continuously on high-performance clustered machines, and hence it would be unrealistic to include additional benchmarks in the case study. The representativeness of the test process used may also impact the generalization of our results. It will be beneficial to complement our case study with industrial case studies in the future. It will also be interesting to study other algorithms and strategies in the future.

6. DISCUSSIONS OF THE APPLICABILITY OF TRADITIONAL TEST ADEQUACY CRITERIA TO PERVASIVE SOFTWARE

A real-life program receives system inputs via system calls or API callback functions. Only trivial programs do not make system or API calls. In pervasive software, environmental contexts constitute part of the system inputs. The same sequence of statements may maintain different pointer values or system constants in the same variables. When the values of these variables are passed to a context-aware system call, the system will respond with an adaptive function. In Android systems, for instance, the values of possible types of wireless channels (such as WiFi, 3G, and Bluetooth) are context values. The choice of specific values will allow the application to adapt to different media for data transmission.

Traditional test adequacy criteria can be applied to ensure the quality of pervasive programs, but since the former do not consider changes due to context-aware system calls, they do not measure whether the adequacy with respect to such contexts has been fulfilled. In theory, one can conduct program analysis of the entire system so that all the data flows from the sensor device level to the application level can be reviewed. In practice, however, the scale of such program analysis is intractable. Lu et al. [2006] takes a projection approach. They abstract all the intermediate code between the sensor and the application, and just model it minimally as an environmental node. In this way, they provide an innovative approach to address the data-flow testing problem for pervasive systems. Unfortunately, the dynamic analysis of context-aware data-flow coverage is still complex in real life despite the abstraction.

In this article, we propose the use of traditional data-flow coverage criteria to test pervasive software, thus bypassing the complexity of intermediate code coverage in Lu et al. [2006]. Context manipulation in terms of diversity is a means to enhance the effectiveness. Compared with an arbitrary change of context, the difference is whether the change affects the context-aware API. In real life, a context change usually causes the program to exercise other paths instead of forcing the same path to use new contexts. Thus, within a program, a context variable behaves like a global variable. However, the scope of influence (in terms of the number of program statements affected) of a context variable is more intensive than that of a local variable.

A context is usually not a simple value precisely representing the captured behavior. Hence, in a context-aware program, such contexts are usually used in a statistical way or portrayed as ranges of values. Thus, a higher discrepancy in the context space will lead to a higher probability in producing failures. This may explain why the three strategies proposed in our article expose failures at different levels of effectiveness, namely, that *CARS-H* performs better than *CARS-E*, which performs better than *CARS-L*.

7. CONCLUSION

This article has proposed the notion of test case substitution in constructing adequate test suites and has studied three strategies to take context changes into account during such construction. The basic idea is to annotate each test case with the amount of its context changes and to select the appropriate test cases based on such amounts whenever choices can be made. The article has also formulated the notion of context diversity as a means to represent the amount of context changes in a test case. We have conducted a multisubject case study to investigate how context changes may be injected into adequate test suites and to what extent such test case substitution can contribute to the effectiveness of the data-flow test adequacy criteria. We have studied six test adequacy criteria, including *PU*, *PUCU*, *CU*, *CUPU*, *AD*, and *AU* as summarized in Section 4.4. We have used three benchmarks with a total of 8,097 lines of code, 30,000 test cases, 959 mutants, and 43,200 adequate test suites for data analysis. To the best of our knowledge, we have presented one of the largest test adequacy experiments on the testing of CPS applications. The experimental results show that the context-aware refined strategies can be successful in changing the context diversity of test suites and can affect the effectiveness of a test adequacy criterion. In particular, we have found in the case study that the *CARS-H* strategy can improve the mean test effectiveness of existing data-flow testing criteria by 10.6%–22.1%, which is significant. On the other hand, *CARS-L* reduces the mean test effectiveness by 2.0%–22.2%, while *CARS-E* can only have a marginal effect. Furthermore, the test suites constructed by *CARS-H* can execute longer paths in a statistically significant way. They provide a clue in understanding the contribution brought by *CARS-H* to the test effectiveness of adequate test suites. Test suites constructed by *CARS-L* tend to execute shorter paths, and *CARS-E* seems to have no significant impact on the execution path lengths of test suites. The experiment also finds that context diversity moderately to strongly correlates with the scope of interactions between CPS applications and their computing environments (in terms of coverage of interacting finite state machines).

In the future, we plan to extend the presented notion of context diversity to model richer information such as complex context representation (e.g., context instances with uncertainty levels), associations among context variables (e.g., a constraint between the location “meeting room” and the activity “watch football”), and different weights of context variables based on their importance to CPS applications. Moreover, we will further study the connection between context changes and program debugging. Context diversity is a black-box metric and it can improve the test effectiveness of adequate test suites. Mutation testing is very time-consuming. As indicated in our previous work [Wang et al. 2010], we are also interested in studying the relationship between mutation analysis and context diversity. We will report our findings in the future.

ACKNOWLEDGMENTS

We are grateful to researchers of the System Research Group of the Department of Computer Science in The University of Hong Kong for their great help in setting up the high-performance cluster environment to complete the experiment.

REFERENCES

R. Alur and M. Yannakakis. 1999. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory* (*CONCUR'99*). Springer, London, UK, 114–129.

P. Ammann and J. Offutt. 2008. *Introduction to Software Testing*, Cambridge University Press, New York, NY.

J. H. Andrews, L. C. Briand, Y. Labiche, and A. Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8, 608–624.

A. Arcuri and L. C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering* (*ICSE'11*). ACM, New York, NY, 1–10.

F. Belina and D. Hogrefe. 1989. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems* 16, 4, 311–341.

A. Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering* (*FOSE'07*) (*in conjunction with the 29th International Conference on Software Engineering* (*ICSE'07*)). IEEE Computer Society, Los Alamitos, CA, 85–103.

D. Binkley and M. Harman. 2004. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering* 30, 11, 715–735.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1980. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL'80*). ACM, New York, NY, 220–233.

C. Chen, C. Ye, and H.-A. Jacobsen. 2011. Hybrid context inconsistency resolution for context-aware services. In *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications* (*PerCom'11*). IEEE Computer Society, Los Alamitos, CA, 10–19.

T. Y. Chen and R. G. Merkel. 2008. An upper bound on software testing effectiveness. *ACM Transactions on Software Engineering and Methodology* 17, 3, 16:1–16:27.

B. Diaz-Agudo, P. A. Gonzalez-Calero, J. A. Recio-Garcia, and A. A. Sanchez-Ruiz-Granados. 2007. Building CBR systems with jCOLIBRI. *Science of Computer Programming* 69, 1–3, 68–75.

V. D'Silva, D. Kroening, and G. Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7, 1165–1178.

J. Edvardsson. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linöping* (*ECSEL'99*). Linöping, Sweden, 21–28.

G. Forney, Jr. 1966. Generalized minimum distance decoding. *IEEE Transactions on Information Theory* 12, 2, 125–131.

P. G. Frankl and S. N. Weiss. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering* 19, 8, 774–787.

P. G. Frankl and E. J. Weyuker. 1988. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 14, 10, 1483–1498.

R. W. Hamming. 1950. Error detecting and error correcting codes. *Bell System Technical Journal* 29, 1, 147–160.

M. Harder, J. Mellen, and M. D. Ernst. 2003. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering* (*ICSE'03*). IEEE Computer Society, Los Alamitos, CA, 60–71.

M. Harman and P. McMinn. 2010. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2, 226–247.

M. M. Hassan and J. H. Andrews. 2013. Comparing multi-point stride coverage and dataflow coverage. In *Proceedings of the 2013 International Conference on Software Engineering* (*ICSE'13*). IEEE, Piscataway, NJ, 172–181.

M. P. E. Heimdahl and D. George. 2004. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering* (*ASE'04*). IEEE Computer Society, Los Alamitos, CA, 176–185.

M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (*ICSE'94*). IEEE Computer Society, Los Alamitos, CA, 191–200.

D. Jeffrey and N. Gupta. 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering* 33, 2, 108–123.

N. H. Kacem, A. H. Kacem, and K. Drira. 2009. A formal model of a multi-step coordination protocol for self-adaptive software using coloured Petri nets. *International Journal of Computing and Information Sciences* 7, 1.

G. M. Kapfhammer and M. L. Soffa. 2003. A family of test adequacy criteria for database-driven applications. In *Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC'03/FSE-11*). ACM, New York, NY, 98–107.

G. Kastrinis and Y. Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'13*). ACM, New York, NY, 423–434.

Z. Lai, S. C. Cheung, and W. K. Chan. 2008. Inter-context control-flow and data-flow test adequacy criteria for nesC applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT'08/FSE-16*). ACM, New York, NY, 94–104.

D. Leon and A. Podgurski. 2003. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering* (*ISSRE'03*). IEEE Computer Society, Los Alamitos, CA, 442–453.

O. Lhoták and K.-C. A. Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL'11*). ACM, New York, NY, 3–16.

J.-W. Lin and C.-Y. Huang. 2009. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology* 51, 4, 679–690.

Y. Liu, C. Xu, and S. C. Cheung. 2013. AFChecker: Effective model checking for context-aware adaptive applications. *Journal of Systems and Software* 86, 3, 854–867.

H. Lu, W. K. Chan, and T. H. Tse. 2006. Testing context-aware middleware-centric programs: A data flow approach and an RFID-based experimentation. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT'06/FSE-14*). ACM, New York, NY, 242–252.

H. Lu, W. K. Chan, and T. H. Tse. 2008. Testing pervasive software in the presence of context inconsistency resolution services. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE'08*). ACM, New York, NY, 61–70.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation* (*PLDI'05*). ACM, New York, NY, 190–200.

Y.-S. Ma, J. Offutt, and Y.-R. Kwon. 2006. MuJava: A mutation system for Java. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE'06*). ACM, New York, NY, 827–830.

L. Mei, W. K. Chan, and T. H. Tse. 2008. Data flow testing of service-oriented workflow applications. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE'08*). ACM, New York, NY, 371–380.

J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. 2005. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE'05*). ACM, New York, NY, 156–165.

A. L. Murphy, G. P. Picco, and G.-C. Roman. 2006. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology* 15, 3, 279–328.

L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil. 2004. LANDMARC: Indoor location sensing using active RFID. *ACM Wireless Networks* 10, 6, 701–710.

J. Offutt, J. Pan, K. Tewary, and T. Zhang. 1996. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience* 26, 2, 165–176.

K. Pearson. 1920. Notes on the history of correlation. *Biometrika* 13, 1, 25–45.

G.-C. Roman, P. J. McCann, and J. Y. Plun. 1997. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology* 6, 3, 250–282.

D. Salber, A. K. Dey, and G. D. Abowd. 1999. The context toolkit: Aiding the development of context-enabled applications. In *The CHI is the Limit: Proceeding of the CHI'99 Conference on Human Factors in Computing Systems*. ACM, New York, NY, 434–441.

M. Sama, S. G. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang. 2010. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering* 36, 5, 644–661.

R. Santelices and M. J. Harrold. 2007. Efficiently monitoring data-flow test coverage. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE'07*). ACM, New York, NY, 343–352.

T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. 2004. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC'04*), Vol. 1. IEEE Computer Society, Los Alamitos, CA, 458–465.

J. von Ronne 1999. Test Suite Minimization: An Empirical Investigation. BSCS Thesis, Oregon State

University, Corvallis, OR.

H. Wang and W. K. Chan. 2009. Weaving context sensitivity into test suite construction. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering* (*ASE'09*). IEEE Computer Society, Los Alamitos, CA, 610–614.

H. Wang, K. Zhai, and T. H. Tse. 2010. Correlating context-awareness and mutation analysis for pervasive computing systems. In *Proceedings of the 10th International Conference on Quality Software* (*QSIC'10*). IEEE Computer Society, Los Alamitos, CA, 151–160.

Z. Wang, S. G. Elbaum, and D. S. Rosenblum. 2007. Automated generation of context-aware tests. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE'07*). IEEE Computer Society, Los Alamitos, CA, 406–415.

E. J. Weyuker. 1990. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering* 16, 2, 121–128.

E. J. Weyuker and T. J. Ostrand. 1980. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering* 6, 3, 236–246.

C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. 2008. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In *Proceedings of the 28th International Conference on Distributed Computing Systems* (*ICDCS'08*). IEEE Computer Society, Los Alamitos, CA, 713–721.

C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. 2010. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology* 19, 3, 9:1–9:61.

Q. Yang, J. J. Li, and D. Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test* (*AST'06*). ACM, New York, NY, 99–103.

Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu. 2013. Formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments. *IEEE Transactions on Parallel and Distributed Systems* 24, 8, 1546–1555.

K. Zhai, B. Jiang, and W. K. Chan. 2014. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *IEEE Transactions on Services Computing* 7, 1, 54–67.

K. Zhai, B. Jiang, W. K. Chan, and T. H. Tse. 2010. Taking advantage of service selection: A study on the testing of location-based web services through test case prioritization. In *Proceedings of the IEEE International Conference on Web Services* (*ICWS'10*). IEEE Computer Society, Los Alamitos, CA, 211–218.

J. Zhang and B. H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE'06*). ACM, New York, NY, 371–380.

J. Zhang, H. J. Goldsby, and B. H. C. Cheng. 2009. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development* (*AOSD'09*). ACM, New York, NY, 161–172.

H. Zhu, P. A. V. Hall, and J. H. R. May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys* 29, 4, 366–427.

Received October 2012; revised December 2013; accepted January 2014

Appendix to: Improving the Effectiveness of Testing Pervasive Software via Context Diversity

HUAI WANG, The University of Hong Kong
W. K. CHAN, City University of Hong Kong
T. H. TSE, The University of Hong Kong

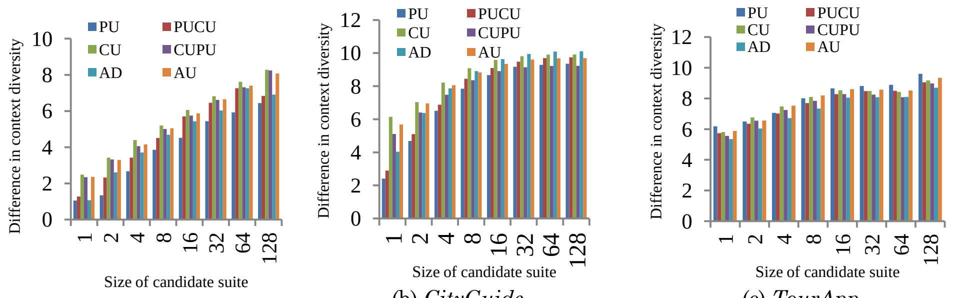
Appendix A

This appendix reports to what extent context diversity can be incorporated into adequate test suites through different strategies. For every benchmark, we have computed the mean context diversity of all the adequate test suites for each combination of adequacy criterion, test suite construction strategy, and size of candidate test suite.

The results for the respective benchmarks are shown in Figures A1, A2, and A3. In each bar chart, the x-axis represents the size of the candidate test suite. Suppose y_1 denotes the mean context diversity of test suites constructed by the baseline strategy *BS* and y_2 denotes that constructed by a context-aware refined strategy *CARS-H*, *CARS-L*, or *CARS-E*. Then, the y-axis represents $y_2 - y_1$, which measures the improvement in mean context diversity of the test suites constructed by a context-aware-refined strategy over that constructed by the baseline strategy. A positive (negative, respectively) bar means that the test suites constructed by the corresponding strategy exhibit higher (lower, respectively) context diversity than that constructed by *BS*.

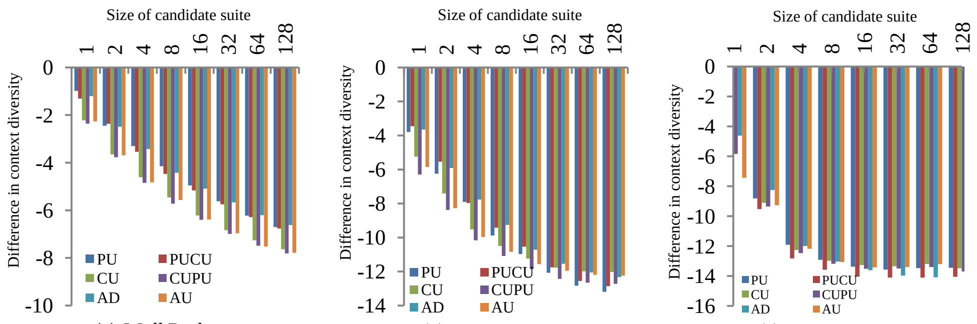
All the bars for the *CARS-H* strategy in Figure A1 are above the x-axis, which indicates that the test suites constructed by *CARS-H* consistently have higher context diversity than those constructed by *BS* strategy regardless of benchmarks, test criteria, and sizes of candidate test suites. The bars of *CARS-H* for larger candidate test suites are longer than those for smaller candidate test suites. It indicates that higher context diversity can be achieved by larger candidate test suites. This observation agrees with the design of *CARS-H* because every test case substitution made by *CARS-H* increases the context diversity of the adequate test suite being constructed, and a larger candidate test suite should provide more trials for a successful substitution. On the other hand, all the bars for the *CARS-L* strategy in Figure A2 are below the x-axis. We find that the difference in context diversity resulting from the use of *CARS-H* and *CARS-L* on each adequacy criterion can be 50%−90% when the size of a candidate test suite is relatively large (such as $k = 32$ or higher). The finding shows that it is feasible to significantly modify the context diversity of an adequate test suite.

We also find that the lengths of the bars for *CARS-L* in Figure A2 tend to be longer than those for *CARS-H* in Figure A1. The result indicates that it can be easier to reduce the context diversity of an adequate test suite than to increase it. Since the two strategies are designed to be symmetric, the asymmetric result thus obtained is interesting. The underlying reason is still unclear. We will leave the explanation to a future study.



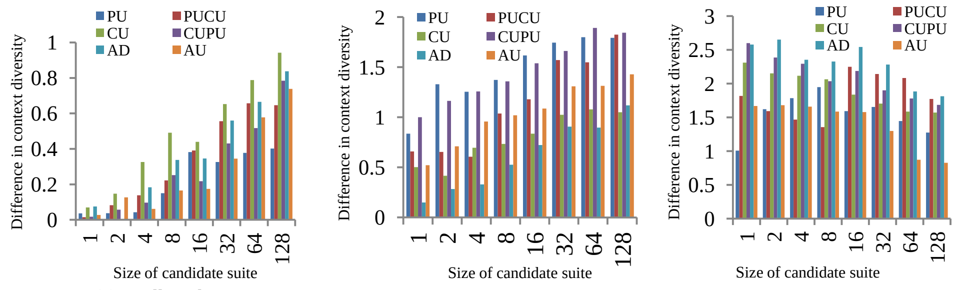
(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A1. Differences in context diversity between *CARS-H* and *BS*.



(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A2. Differences in context diversity between *CARS-L* and *BS*.



(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A3. Differences in context diversity between *CARS-E* and *BS*.

Although all the bars for *CARS-E* in Figure A3 are above the *x*-axis, they are much shorter than the corresponding bars in both Figures A1 and A2. Compared with the context diversity of the adequate test suites produced by *BS*, the lengths of the bars for *CARS-E* in Figure A3 only indicate a difference of 7%–27%, which is noticeable but less significant than *CARS-H* and *CARS-L* in general. The finding indicates that the idea of evenly spreading test cases in the context diversity dimension exhibits less effectiveness than *CARS-H* and *CARS-L* in the case study.

Table A1. Bonferroni Multiple Comparisons of Context Diversity of Test Suites Constructed by Different Strategies for *WalkPath*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	17–21	18–22	**22–25**	**25–28**	**27–30**	**29–32**	**31–33**	**32–35**
	PUCU	17–21	**21–23**	**23–26**	**26–29**	**29–32**	**31–34**	**33–36**	**32–35**
	CU	**21–24**	**23–26**	**26–29**	**28–31**	**30–33**	**32–35**	**34–37**	**36–38**
	CUPU	**21–24**	**23–26**	**26–28**	**28–31**	**30–33**	**32–35**	**34–37**	**36–38**
	AD	17–20	**21–24**	**24–27**	**26–29**	**28–31**	**30–33**	**33–36**	**32–35**
	AU	**22–24**	**23–26**	**26–29**	**28–31**	**30–33**	**32–35**	**34–37**	**36–38**
CARS-L	PU	15–16	**12–14**	**10–12**	**8–10**	**6–9**	**4–7**	**2–5**	**1–4**
	PUCU	13–15	**11–13**	**9–11**	**6–9**	**4–7**	**3–6**	**1–4**	**1–3**
	CU	**12–14**	**9–11**	**6–9**	**4–7**	**2–5**	**1–3**	**1–2**	**1**
	CUPU	**12–14**	**9–11**	**6–9**	**3–7**	**2–5**	**1–3**	**1–2**	**1**
	AD	13–15	**10–13**	**9–11**	**6–9**	**4–8**	**3–6**	**1–4**	**1–3**
	AU	**12–14**	**9–11**	**6–9**	**4–7**	**2–5**	**1–3**	**1–2**	**1**
CARS-E	PU	16–18	16–18	16–18	16–18	16–19	16–19	16–19	17–19
	PUCU	15–17	16–17	16–17	16–17	16–18	16–18	16–19	16–19
	CU	16–17	16–17	16–18	16–19	16–18	17–19	17–20	17–20
	CUPU	16–18	16–18	16–18	16–18	16–18	17–19	17–19	17–20
	AD	16–17	15–17	16–17	16–18	16–18	16–19	16–19	17–20
	AU	16–18	16–18	16–18	16–18	16–18	16–19	17–20	17–20
BS	PU	16–18	16–18	16–18	16–18	16–18	16–18	16–18	16–18
	PUCU	15–17	15–17	15–17	15–17	15–17	15–17	15–17	15–17
	CU	16–17	16–17	16–17	16–17	16–17	16–17	16–17	16–17
	CUPU	16–18	16–18	16–18	16–18	16–18	16–18	16–18	16–18
	AD	15–17	15–17	15–17	15–17	15–17	15–17	15–17	15–17
	AU	16–18	16–18	16–18	16–18	16–18	16–18	16–18	16–18

Table A2. Bonferroni Multiple Comparisons of Context Diversity of Test Suites Constructed by Different Strategies for *CityGuide*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	14–18	**20–22**	**22–24**	**24–26**	**24–28**	**25–29**	**25–29**	**25–29**
	PUCU	14–20	**20–22**	**22–24**	**24–27**	**24–28**	**25–29**	**25–29**	**25–29**
	CU	**21–23**	**22–24**	**23–26**	**24–27**	**25–28**	**25–28**	**25–29**	**25–29**
	CUPU	**21–22**	**22–24**	**23–25**	**24–27**	**25–28**	**25–29**	**25–29**	**25–29**
	AD	**17–21**	**21–23**	**22–25**	**24–27**	**24–28**	**25–29**	**25–29**	**25–29**
	AU	**21–23**	**22–24**	**23–26**	**24–27**	**25–28**	**25–29**	**25–29**	**25–29**
CARS-L	PU	**11–13**	**8–11**	**6–9**	**3–7**	**1–5**	**1–3**	**1**	**1**
	PUCU	**11–13**	**9–11**	**6–8**	**3–7**	**1–5**	**1–3**	**1**	**1**
	CU	**9–11**	**6–9**	**2–6**	**1–4**	**1–3**	**1–2**	**1–2**	**1–2**
	CUPU	**8–10**	**6–8**	**3–6**	**1–4**	**1–3**	**1–2**	**1–2**	**1–2**
	AD	**10–12**	**8–10**	**5–8**	**3–6**	**1–4**	**1–2**	**1**	**1**
	AU	**8–11**	**5–8**	**2–6**	**1–4**	**1–3**	**1–2**	**1–2**	**1–2**
CARS-E	PU	14–17	14–18	14–17	14–18	14–18	15–18	15–19	15–19
	PUCU	14–16	14–16	14–16	14–16	14–17	14–17	14–17	14–18
	CU	14–15	14–15	14–15	14–15	14–15	14–16	14–16	14–16
	CUPU	14–17	14–17	14–17	14–18	14–18	14–18	15–19	15–19
	AD	14	14	14	14–15	14–15	14–15	14–15	14–16
	AU	14–15	14–16	14–16	14–16	14–16	14–17	14–17	14–17
BS	PU	14–15	14–15	14–15	14–15	14–15	14–15	14–15	14–15
	PUCU	14–15	14–15	14–15	14–15	14–15	14–15	14–15	14–15
	CU	14	14	14	14	14	14	14	14
	CUPU	14–15	14–15	14–15	14–15	14–15	14–15	14–15	14–15
	AD	14	14	14	14	14	14	14	14
	AU	14	14	14	14	14	14	14	14

Table A3. Bonferroni Multiple Comparisons of Context Diversity of Test Suites Constructed by Different Strategies for *TourApp*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	**14–16**	**15–16**	**16–17**	**16–18**	**16–19**	**16–19**	**16–19**	**17–20**
	PUCU	**15–16**	**16–17**	**16–18**	**16–19**	**16–20**	**16–20**	**16–20**	**17–21**
	CU	**14–16**	**15–17**	**16–18**	**16–18**	**16–19**	**16–19**	**16–19**	**16–20**
	CUPU	**14–16**	**15–17**	**16–18**	**16–18**	**16–19**	**16–19**	**16–19**	**16–20**
	AD	**14–16**	**15–17**	**16–18**	**16–18**	**16–19**	**16–20**	**16–20**	**17–20**
	AU	**14–16**	**15–17**	**16–18**	**16–19**	**16–19**	**16–19**	**16–19**	**16–20**
CARS-L	PU	**6–7**	**3–5**	**1–3**	**1**	**1**	**1**	**1**	**1**
	PUCU	**6–8**	**3–5**	**1–3**	**1–2**	**1**	**1**	**1**	**1**
	CU	**5–7**	**3–5**	**1–3**	**1–2**	**1**	**1**	**1**	**1**
	CUPU	**6–7**	**3–5**	**1–3**	**1–2**	**1**	**1**	**1**	**1**
	AD	**7–9**	**5–6**	**1–4**	**1–3**	**1–2**	**1**	**1**	**1**
	AU	**5–6**	**3–5**	**1–3**	**1**	**1**	**1**	**1**	**1**
CARS-E	PU	10–11	10–12	10–12	10–12	10–12	10–12	10–12	10–11
	PUCU	10–13	10–13	10–13	10–12	10–14	10–14	10–14	10–13
	CU	10–13	10–13	10–13	10–13	10–12	10–12	10–12	10–12
	CUPU	10–14	10–14	10–13	10–13	10–13	10–13	10–13	10–13
	AD	10–15	11–15	10–14	10–14	10–14	10–14	10–14	10–13
	AU	10–12	10–12	10–12	10–12	10–12	10–12	10–11	10–11
BS	PU	10	10	10	10	10	10	10	10
	PUCU	10–11	10–11	10–11	10–11	10–11	10–11	10–11	10–11
	CU	10	10	10	10	10	10	10	10
	CUPU	10	10	10	10	10	10	10	10
	AD	10–11	10–11	10–11	10–11	10–11	10–11	10–11	10–11
	AU	10	10	10	10	10	10	10	10

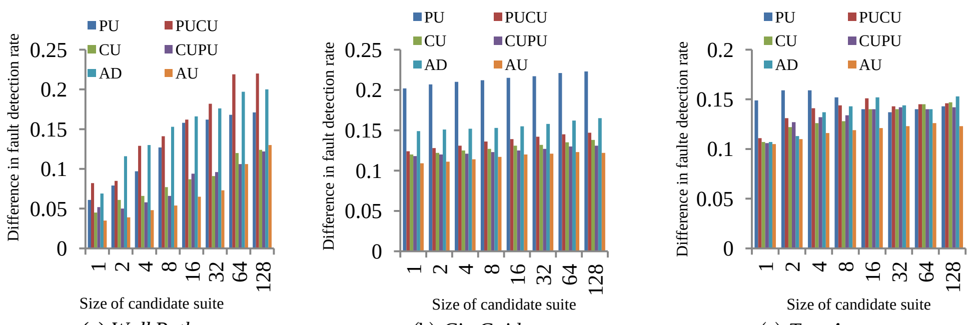
Moreover, for all benchmarks and test criteria in Figure A1, *CARS-H* always brings in the smallest increase in context diversity when $k = 1$. Note that when $k = 1$, the candidate test suite defined in Figure A1 contains only one test case, and the context-aware sampling strategy degrades into the random strategy. However, *CARS-H* may still have a chance to increase context diversity of the test suites by solving tie cases during test suite construction: According to the definition of the *replace*() function in Algorithm 2, the context diversity of the test suite being constructed will be increased when a test case is replaced. The positive bars for $k = 1$ suggests that the tie case solving strategy employed by *CARS-H* (and implemented by the *replace*() function) contributes to an increase in context diversity. When $k > 1$, the candidate test suite contains more than one test case and has more chances to select test cases with higher context diversity. The higher context diversity for $k > 1$ than for $k = 1$ suggests that nontrivial candidate test suites (implemented by the *select*() function) contributes to an increase in context diversity. Similarly, the negative bars for $k = 1$ in Figure A2 suggest that the *replace*() function in *CARS-L* tends to reduce context diversity, and the lower context diversity for $k > 1$ than for $k = 1$ suggests that the *select*() function in *CARS-L* tends to reduce context diversity also.

To investigate whether the difference between various test suite construction strategies and the baseline strategy is significant, we have conducted Bonferroni multiple comparison analysis for every test suite construction strategy. The respective results for *WalkPath*, *CityGuide*, and *TourApp* are presented in Tables A1, A2, and A3. For any candidate test suite size, if the range of numbers in the cell of a refined strategy has an overlap with the range of numbers in the cell of the baseline strategy, then there is no significant difference in the mean context diversity at a confidence level of 95% and thus it fails to reject the null hypothesis that "test suites constructed by a specific context-aware refined strategy share the same context diversity with those constructed by the baseline strategy *BS*." Otherwise, the difference is significant and the corresponding cells reject the null hypothesis. For example, for the test criterion *PU* and the candidate test suite size = 1 in Table A1, test suites constructed by *CARS-H* do not have significantly different context diversity from that constructed by *BS* because the ranges of numbers in their respective cells have an overlap of the numbers 17 and 18. However, for the test criterion *CU*, test suites constructed by *CARS-H* have significantly different context diversity from that constructed by *BS* because the ranges of numbers in their respective cells have no overlap. For ease of recognition, we have highlighted the cells that reject the null hypothesis.

From Tables A1, A2, and A3, we find that regardless of benchmarks and test criteria, both *CARS-H* and *CARS-L* are more likely to reject the null hypothesis. For example, *CARS-H* rejects the null hypothesis in 44, 46, and 48 out of 48 cases for *WalkPath*, *CityGuide*, and *TourApp*, respectively. *CARS-L* can reject the null hypotheses in 45, 48, and 48 out of 48 cases for *WalkPath*, *CityGuide*, and *TourApp*, respectively. In contrast, *CARS-E* consistently fails to reject the null hypothesis for candidate test suites of all sizes. This result validates that both *CARS-H* and *CARS-L* have a significant impact on the context diversity of test suites, and the impact of *CARS-E* to the context diversity of test suites is not statistically significant.

Appendix B

To study the test effectiveness enhanced by context diversity considerations using different strategies, we compare the strategies at the criterion level. Figures A4, A5, and A6 show the changes in test effectiveness between the baseline strategy *BS* and a context-aware refined strategy *CARS-H*, *CARS-L*, and *CARS-E*, respectively. The x-axis of each plot in Figures A4–A6 shows the size of candidate test suites. Suppose y_1 denotes the mean fault detection rate of a criterion (such as *PU*, *PUCU*, *CU*, *CUPU*, *AD*, and *AU*) of the baseline strategy and y_2 denotes that of a context-aware refined strategy *CARS-H*, *CARS-L*, or *CARS-E*. Then, the y-axis represents $y_2 - y_1$, which is the difference between the fault detection rate of a context-aware refined strategy and that of the baseline strategy with respect to a specific adequacy criterion. Note that a positive bar for a specific adequacy criterion indicates that a context-aware refined strategy is more effective than the *BS* strategy in exposing faults.



(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A4. Differences in fault detection rates between *CARS-H* and *BS*.

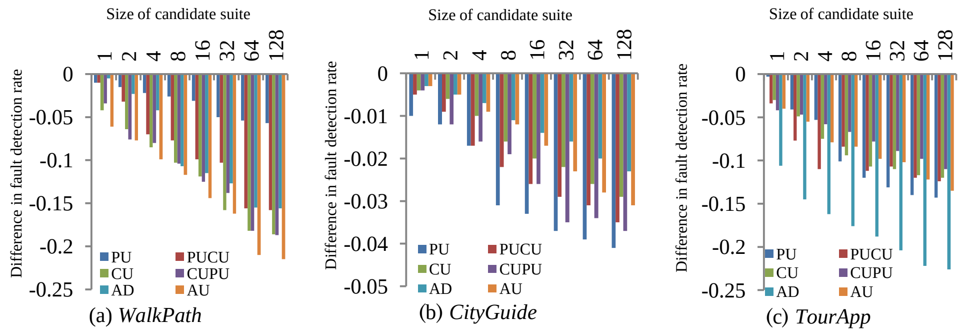


Fig. A5. Differences in fault detection rates between *CARS-L* and *BS*.

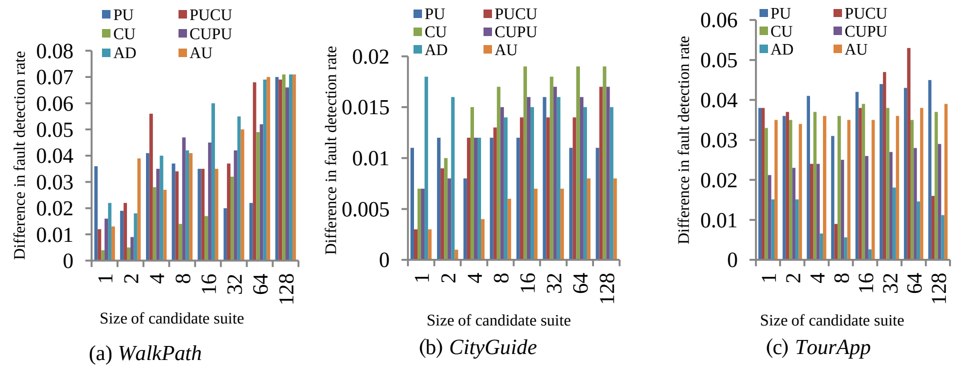


Fig. A6. Differences in fault detection rates between *CARS-E* and *BS*.

When analyzing the mean fault detection rates of various criteria paired with *CARS-H*, we find that all the bars in Figure A4 are positive. This observation implies that an adequacy criterion paired with the *CARS-H* strategy can be more effective than that paired with the baseline strategy *BS* in terms of fault detection rates. For *WalkPath*, for instance, when *k* varies from 1 to 128, *CARS-H* outperforms *BS* by 6.1%–17.1% for *PU*, 8.2%–22.0% for *PUCU*, 4.5%–12.4% for *CU*, 5.2%–10.6% for *CUPU*, 6.9%–20.0% for *AD*, and 3.5%–10.6% for *AU*. Similar observations can be found for *CityGuide* and *TourApp*. In other words, *CARS-H* can consistently improve the test effectiveness in terms of the mean fault detection rate regardless of benchmarks, test criteria, and sizes of candidate test suites. Considering that data-flow testing is by itself a highly effective technique, the result shows that the improvement made by *CARS-H* is significant.

Furthermore, for *CARS-H*, the lengths of bars become longer and longer when the value of *k* increases, and the effect of *k* seems to be saturated at *k* = 64 in the experiment. For example, the maximum difference in test effectiveness between *k* = 64 and *k* = 128 never exceeds 2.4%, 0.3%, and 1.3% for *WalkPath*, *CityGuide*, *TourApp*, respectively, regardless of the adequacy criterion. In particular, when *k* = 64, the improvement of test effectiveness for all criteria induced by *CARS-H* over *BS* is 10.6%–21.9% for *WalkPath*, 12.3%–22.1% for *TourApp*, and 12.6%–14.5% for *CityGuide*. In summary, irrespective of benchmarks and criteria, when *k* = 64, *CARS-H* can bring about an improvement of 10.6%–22.1% in test effectiveness to the baseline strategy in terms of the mean fault detection rate.

In Figure A5, all the bars of *CARS-L* are negative. This suggests that irrespective of benchmarks and test criteria, the *CARS-L* strategy always performs less effectively than the *BS* strategy in terms of achieving the mean fault detection rates. In particular, for all the criteria when k = 64, *CARS-L* is 5.4%–21%, 2.0%–3.9%, and 9.8%–22.2% less effective than *BS* for *WalkPath*, *CityGuide*, and *TourApp*, respectively. That is, regardless of benchmarks and test criteria, *CARS-L* is 2.0%–22.2% less effective than *BS*.

Combining with the result presented in Appendix A, we find that changing the context diversity of an adequate test suite using *CARS-H* and *CARS-L* does correlate with the change in test effectiveness.

CARS-E in Figure A6 results in positive bars, which are shorter than those corresponding to *CARS-H* and *CARS-L*. In particular, for all the test criteria when k = 64, *CARS-E* can be more effective by 2.2%–7.0%, 0.8%–1.9%, and 1.4%–5.3% for *WalkPath*, *CityGuide* and *TourApp*, respectively. The result suggests that *CARS-E* brings in less impact on the test effectiveness of test suites than *CARS-H* or *CARS-L*.

Table A4. Bonferroni Multiple Comparisons of Fault Detection Rates of Test Suites Constructed by Different Strategies for *WalkPath*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	**2**	**4**	**8**	**16**	**32**	**64**	**128**
CARS-H	PU	3–5	3–5	**4–6**	**5–7**	**6–8**	**7–8**	**7–8**	**7–8**
	PUCU	**10–12**	**10–12**	**12–14**	**12–14**	**13–15**	**14–16**	**15–17**	**15–17**
	CU	17–18	17–18	17–18	17–18	17–18	**18–19**	**19–21**	**19–21**
	CUPU	17–18	17–18	17–18	17–18	**18–20**	**18–20**	**18–20**	**19–21**
	AD	9–11	**11–13**	**12–14**	**13–15**	**13–15**	**14–16**	**14–16**	**14–17**
	AU	17–18	17–18	17–18	17–18	17–18	17–18	**18–21**	**19–22**
CARS-L	PU	1–2	1–2	1–2	1–2	1–2	1	1	1
	PUCU	7–9	7–8	5–8	**5–7**	**4–6**	**4–6**	**2–4**	**2–4**
	CU	13–15	12–15	**12–13**	**11–13**	**10–12**	**9–11**	**8–10**	**8–10**
	CUPU	14–16	12–15	**12–14**	**11–13**	**10–12**	**10–12**	**8–10**	**8–10**
	AD	7–9	7–8	6–8	**4–6**	**4–5**	**3–5**	**2–4**	**2–4**
	AU	13–15	12–15	**12–13**	**11–13**	**10–12**	**9–11**	**8–9**	**8–9**
CARS-E	PU	2–4	3–5	3–4	3–5	3–5	3–5	3–5	3–4
	PUCU	9–10	9–11	9–11	9–10	9–10	9–11	9–10	9–11
	CU	17–18	17–18	17–18	18	18–19	18–19	18–19	18–19
	CUPU	17–18	17–18	18	17–18	18–19	18–19	18–20	18–19
	AD	8–10	8–10	9–11	9–11	9–10	9–10	9–11	9–11
	AU	17–18	18	17–18	17–18	18–19	18	18–20	18–19
BS	PU	1–3	1–3	1–3	1–3	1–3	1–3	1–3	1–3
	PUCU	8–9	8–9	8–9	8–9	8–9	8–9	8–9	8–9
	CU	15–17	15–17	15–17	15–17	15–17	15–17	15–17	15–17
	CUPU	15–17	15–17	15–17	15–17	15–17	15–17	15–17	15–17
	AD	8–9	8–9	8–9	8–9	8–9	8–9	8–9	8–9
	AU	15–17	15–17	15–17	15–17	15–17	15–17	15–17	15–17

Table A5. Bonferroni Multiple Comparisons of Fault Detection Rates of Test Suites Constructed by Different Strategies for *CityGuide*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	**2**	**4**	**8**	**16**	**32**	**64**	**128**
CARS-H	PU	**11**	**11**	**11**	**11**	**11**	**11–12**	**11–12**	**11–12**
	PUCU	**11**	**11**	**11**	**11–12**	**11–12**	**11–12**	**11–12**	**11–13**
	CU	**11–12**	**11–12**	**11–12**	**11–12**	**11–13**	**11–13**	**11–13**	**11–13**
	CUPU	**11–12**	**11–12**	**11–12**	**11–12**	**11–12**	**11–12**	**11–13**	**11–13**
	AD	**11–12**	**11–12**	**11–12**	**11–12**	**11–12**	**11–12**	**11–12**	**11–13**
	AU	**11–12**	**11–12**	**11–12**	**11–12**	**11–13**	**11–13**	**11–13**	**11–13**
CARS-L	PU	1–2	1–2	1–2	1	1	1	1	1
	PUCU	5–7	5–7	5–6	4–6	4–5	4–5	4–5	3–5
	CU	6–8	5–8	5–8	5–7	5–7	5–7	5–7	5–6
	CUPU	6–8	5–8	5–7	5–7	5–7	5–6	5–6	4–6
	AD	4–6	4–6	4–6	4–5	4–5	4–5	3–5	3–5
	AU	6–9	6–9	6–9	6–8	5–8	5–8	5–7	5–7
CARS-E	PU	2–3	2–3	1–3	2–3	2–3	2–4	2–3	2–3
	PUCU	5–7	5–8	6–8	6–8	6–8	6–8	6–8	6–8
	CU	6–9	6–9	7–9	7–10	7–10	7–10	7–10	7–10
	CUPU	6–9	6–9	7–9	7–10	7–10	7–10	7–10	7–10
	AD	5–7	5–7	5–7	5–7	5–7	5–7	5–7	5–7
	AU	7–9	7–9	7–9	7–10	7–10	7–10	7–10	7–10
BS	PU	1–3	1–3	1–3	1–3	1–3	1–3	1–3	1–3
	PUCU	5–7	5–7	5–7	5–7	5–7	5–7	5–7	5–7
	CU	6–8	6–8	6–8	6–8	6–8	6–8	6–8	6–8
	CUPU	6–9	6–9	6–9	6–9	6–9	6–9	6–9	6–9
	AD	5–6	5–6	5–6	5–6	5–6	5–6	5–6	5–6
	AU	7–9	7–9	7–9	7–9	7–9	7–9	7–9	7–9

Table A6. Bonferroni Multiple Comparisons of Fault Detection Rates of Test Suites Constructed by Different Strategies for *TourApp*

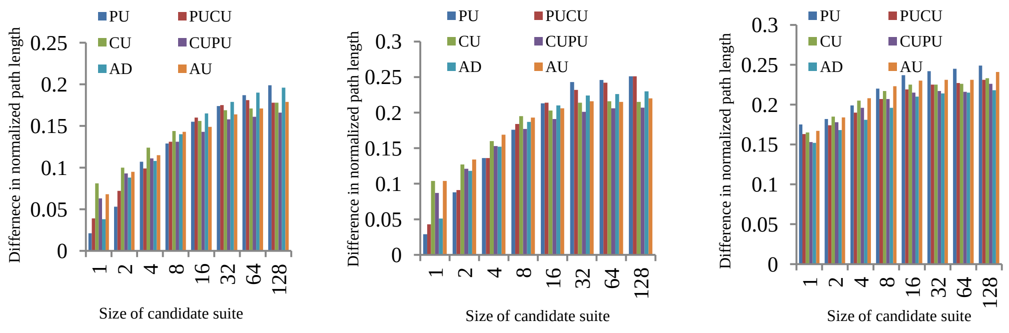
Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	**14–18**	**15–18**	**15–18**	**14–18**	**14–17**	**13–17**	**14–17**	**14–17**
	PUCU	**18–22**	**19–23**	**20–24**	**20–24**	**20–24**	**20–24**	**20–24**	**20–24**
	CU	**21–25**	**22–26**	**23–26**	**23–26**	**24–27**	**24–27**	**24–27**	**24–27**
	CUPU	**22–25**	**23–26**	**23–26**	**23–27**	**24–27**	**24–27**	**24–27**	**24–27**
	AD	**14–17**	**16–20**	**18–21**	**18–22**	**19–22**	**18–22**	**18–21**	**19–22**
	AU	**22–26**	**23–27**	**24–27**	**24–27**	**24–27**	**24–28**	**25–28**	**24–28**
CARS-L	PU	6–9	4–6	3–6	**1–4**	**1–3**	**1–2**	**1–2**	**1–2**
	PUCU	10–13	**7–10**	**5–9**	**6–10**	**5–8**	**5–9**	**5–8**	**5–8**
	CU	13–17	12–16	**10–14**	**10–13**	**9–12**	**9–12**	**8–11**	**8–11**
	CUPU	13–16	12–16	**12–14**	**11–14**	**10–14**	**10–13**	**10–13**	**9–12**
	AD	**4–7**	**2–5**	**1–4**	**1–3**	**1–3**	**1–2**	**1**	**1**
	AU	14–18	13–17	**12–15**	**12–15**	**11–14**	**10–14**	**10–13**	**9–12**
CARS-E	PU	8–11	8–11	8–11	7–11	8–11	8–11	8–11	8–12
	PUCU	14–17	13–17	13–16	12–15	14–17	14–18	15–18	12–16
	CU	17–20	17–21	17–21	17–21	17–21	17–21	17–21	17–21
	CUPU	17–20	17–20	17–20	17–20	17–20	17–20	17–20	17–20
	AD	10–14	10–14	10–13	10–13	10–13	10–14	10–14	10–14
	AU	19–22	19–22	19–22	19–22	19–22	19–22	19–23	19–23
BS	PU	6–9	6–9	6–9	6–9	6–9	6–9	6–9	6–9
	PUCU	11–15	11–15	11–15	11–15	11–15	11–15	11–15	11–15
	CU	15–18	15–18	15–18	15–18	15–18	15–18	15–18	15–18
	CUPU	15–19	15–19	15–19	15–19	15–19	15–19	15–19	15–19
	AD	10–13	10–13	10–13	10–13	10–13	10–13	10–13	10–13
	AU	17–20	17–20	17–20	17–20	17–20	17–20	17–20	17–20

To find out whether the results are statistically significant, we have also conducted Bonferroni multiple comparison analysis for every test suite construction strategy. The respective results for *WalkPath*, *CityGuide*, and *TourApp* are presented in Tables A4, A5, and A6, which can be interpreted similarly to Tables A1, A2, and A3, where the highlighted cells indicate the rejection of the null hypothesis that "test suites constructed by the context-aware refined strategy share the same fault detection rate as the baseline strategy *BS*".

From Tables A4, A5, and A6, we observe that regardless of benchmarks and test criteria, *CARS-H* is likely to reject the null hypothesis. For example, *CARS-H* rejects the null hypothesis in 30, 48, and 48 out of 48 cases for *WalkPath*, *CityGuide*, and *TourApp*, respectively. This result confirms that *CARS-H* can outperform the *BS* strategy significantly in terms of fault detection rates achieved by test suites with a specific size and coverage. In contrast, *CARS-L* deteriorates the effectiveness of test suites significantly. For example, *CARS-L* rejects the null hypothesis in 28 and 38 out of 48 cases for *WalkPath* and *TourApp*. Quite a number of outliers exist for *CityGuide*: *CARS-L* fails to reject the null hypothesis for all the cases. As shown in Table V in Section 5.1, owing to the low fault detection rates achieved by test suites constructed by the baseline strategy *BS*, *CARS-L* cannot deteriorate the test effectiveness further. Compared with *CARS-H* and *CARS-L*, *CARS-E* cannot be significantly distinguished from *BS*. For example, *CARS-E* fails to reject the null hypothesis in all the 48 cases for the three benchmarks.

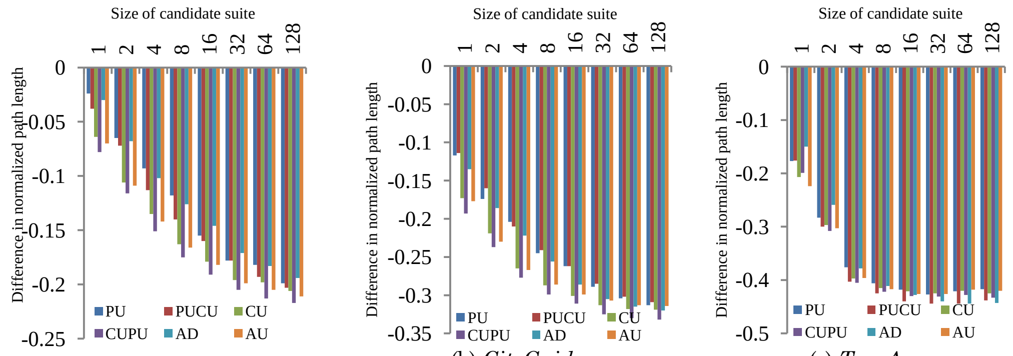
Appendix C

In this appendix, we report to what extent the context diversity of test suites correlates with the execution of test suites in terms of the normalized execution path length expressed as the number of statements executed. We normalize the length of an execution path as the ratio of the length of that path to the length of the longest execution path executed by the test cases in the test pool. Figures A7, A8, and A9 show the changes in the normalized execution path lengths between the baseline strategy *BS* and a context-aware refined strategy (*CARS-H*, *CARS-L*, and *CARS-E*, respectively). The x-axis of each plot in Figures A7–A9 shows the size of the candidate test suite. Suppose $y1$ denotes the mean normalized execution path length of a criterion (such as *PU*, *PUCU*, *CU*, *CUPU*, *AD*, and *AU*) of the baseline strategy and $y2$ denotes that of a context-aware refined strategy *CARS-H*, *CARS-L*, or *CARS-E*. Then, the y-axis represents $y2 - y1$, which is the difference between the normalized execution path length of a context-aware refined strategy and that of the baseline strategy with respect to a specific adequacy criterion. Note that a positive bar for a specific adequacy criterion indicates that a context-aware refined strategy exercises a longer path than the *BS* strategy.



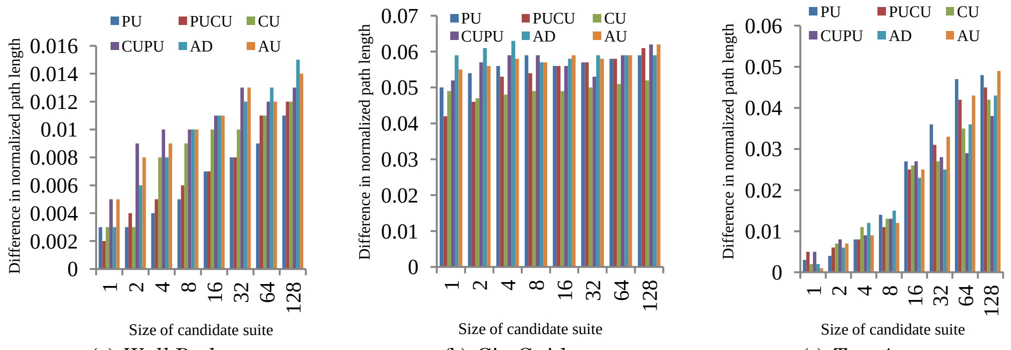
(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A7. Differences in normalized execution path lengths between *CARS-H* and *BS*.



(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A8. Differences in normalized execution path lengths between *CARS-L* and *BS*.



(a) *WalkPath* (b) *CityGuide* (c) *TourApp*

Fig. A9. Differences in normalized execution path lengths between *CARS-E* and *BS*.

For each normalized execution path length shown in Figure A7, *CARS-H* always results in a positive bar, which indicates that test suites constructed by *CARS-H* consistently exercise more statements than those constructed by *BS*. For example, irrespective of test criteria, when varying *k* from 1 to 128, the minimum and maximum numbers of changes in the normalized execution path lengths brought by *CARS-H* to *BS* are 2.1% and 19.9% for *WalkPath*, 2.9% and 25.1% for *CityGuide*, and 15.2% and 24.9% for *TourApp*, respectively. That is, the test suites constructed by

CARS-H exercise 2.1%–25.1% longer paths than those constructed by *BS*. In Figure A8, we observe that *CARS-L* always executes shorter paths than the baseline strategy *BS*. *CARS-E* in Figure A9 is associated with positive bars but the lengths are much shorter than those corresponding to *CARS-H*, which indicates that *CARS-E* has much less effect than *CARS-H* in forcing test suites to traverse longer paths.

To investigate whether the changes due to various strategies over the *BS* strategy are attributed to chance, we have conducted Bonferroni multiple comparison analysis for every strategy. The results for *WalkPath*, *CityGuide*, and *TourApp* are presented in Tables A7, A8, and A9, respectively, where the highlighted cells reject the null hypothesis that "test suites constructed by a context-aware refined strategy traverse execution paths of the same lengths as test suites constructed by the baseline strategy *BS*".

We observe from Tables A7–A9 that both *CARS-H* and *CARS-L* tend to reject the null hypothesis and *CARS-E* fails to reject the null hypothesis regardless of benchmarks, test criteria, and sizes of candidate test suites. For example, *CARS-H* rejects the null hypothesis in 44, 44, and 48 out of 48 cases for *WalkPath*, *CityGuide*, and *TourApp*, *CARS-L* rejects the null hypothesis in 39, 48, and 48 out of 48 cases for the three benchmarks, and *CARS-E* fails to reject the null hypothesis in all 48 cases for the three benchmarks, respectively. The results imply that test suites constructed by *CARS-H* execute significantly more statements (in the statistical sense) than those constructed by the *BS* strategy, test suites constructed by *CARS-L* execute significantly fewer statements, and test suites constructed by *CARS-E* traverse a similar number of statements.

Table A7. Bonferroni Multiple Comparisons of Normalized Execution Path Lengths Exercised by Test Suites Constructed by Different Strategies for *WalkPath*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	8–12	10–14	**12–16**	**13–17**	**15–19**	**15–20**	**16–21**	**17–21**
	PUCU	9–14	**11–15**	**12–16**	**14–18**	**15–19**	**16–20**	**16–21**	**16–21**
	CU	**13–16**	**13–17**	**14–18**	**15–19**	**15–20**	**16–21**	**16–21**	**16–21**
	CUPU	**13–15**	**13–17**	**14–18**	**15–19**	**15–20**	**16–21**	**16–21**	**16–21**
	AD	9–13	**11–15**	**12–16**	**14–18**	**15–19**	**15–20**	**16–21**	**16–21**
	AU	**13–15**	**13–17**	**14–18**	**15–19**	**15–19**	**16–20**	**16–21**	**17–21**
CARS-L	PU	6–10	4–8	**2–6**	**1–5**	**1–3**	**1–2**	**1–2**	**1**
	PUCU	6–10	4–8	**1–6**	**1–5**	**1–4**	**1–3**	**1–2**	**1**
	CU	5–9	**2–7**	**1–5**	**1–4**	**1–3**	**1–2**	**1–2**	**1**
	CUPU	5–9	**2–7**	**1–5**	**1–4**	**1–3**	**1–2**	**1**	**1**
	AD	6–10	4–8	**1–6**	**1–5**	**1–4**	**1–2**	**1–2**	**1**
	AU	5–9	**2–7**	**1–5**	**1–4**	**1–3**	**1–2**	**1**	**1**
CARS-E	PU	7–11	7–11	7–11	7–11	7–11	7–11	7–11	8–11
	PUCU	8–11	8–11	8–11	8–12	8–12	8–12	8–12	8–12
	CU	8–12	8–12	8–12	8–12	8–12	8–12	8–12	8–12
	CUPU	9–13	9–13	9–13	9–13	9–13	9–13	9–13	9–13
	AD	7–11	7–11	7–11	8–11	8–11	8–11	8–11	8–12
	AU	8–12	8–12	8–12	8–13	8–13	9–13	8–13	9–13
BS	PU	7–11	7–11	7–11	7–11	7–11	7–11	7–11	7–11
	PUCU	7–11	7–11	7–11	7–11	7–11	7–11	7–11	7–11
	CU	8–12	8–12	8–12	8–12	8–12	8–12	8–12	8–12
	CUPU	8–12	8–12	8–12	8–12	8–12	8–12	8–12	8–12
	AD	7–11	7–11	7–11	7–11	7–11	7–11	7–11	7–11
	AU	8–12	8–12	8–12	8–12	8–12	8–12	8–12	8–12

Table A8. Bonferroni Multiple Comparisons of Normalized Execution Path Lengths Exercised by Test Suites Constructed by Different Strategies for *CityGuide*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	7–8	8–10	**9–11**	**11–13**	**12–14**	**13–15**	**13–15**	**13–15**
	PUCU	7–8	8–10	**9–11**	**11–13**	**12–14**	**13–15**	**13–15**	**13–15**
	CU	**9–11**	**10–12**	**11–13**	**12–14**	**12–14**	**12–14**	**13–15**	**12–15**
	CUPU	8–11	**10–12**	**11–13**	**12–14**	**12–14**	**12–14**	**13–15**	**13–15**
	AD	7–9	**9–11**	**10–12**	**11–13**	**12–14**	**13–15**	**13–15**	**13–15**
	AU	**9–11**	**10–12**	**11–13**	**12–14**	**12–14**	**12–14**	**12–14**	**13–15**
CARS-L	PU	**5–6**	**3–5**	**2–4**	**1–3**	**1–3**	**1–2**	**1**	**1**
	PUCU	**5–6**	**4–5**	**2–4**	**1–3**	**1–2**	**1–2**	**1**	**1**
	CU	**4–5**	**2–4**	**1–3**	**1–2**	**1–2**	**1**	**1**	**1**
	CUPU	**4–5**	**2–4**	**1–3**	**1–2**	**1–2**	**1**	**1**	**1**
	AD	**5–6**	**3–5**	**2–4**	**1–3**	**1–2**	**1**	**1**	**1**
	AU	**4–5**	**2–4**	**1–3**	**1–2**	**1–2**	**1**	**1**	**1**
CARS-E	PU	7–9	7–9	7–9	7–9	7–9	7–9	7–9	7–9
	PUCU	7–8	7–8	7–8	7–9	7–9	7–9	7–9	7–9
	CU	7–9	7–9	7–9	7–9	7–9	7–9	7–9	7–9
	CUPU	7–10	7–10	7–10	7–10	7–10	7–10	7–10	8–10
	AD	7–9	7–9	7–9	7–9	7–9	7–9	7–9	7–9
	AU	7–9	7–9	7–9	7–9	7–9	7–9	7–9	7–9
BS	PU	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8
	PUCU	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8
	CU	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8
	CUPU	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8
	AD	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8
	AU	7–8	7–8	7–8	7–8	7–8	7–8	7–8	7–8

Table A9. Bonferroni Multiple Comparisons of Normalized Execution Path Lengths Exercised by Test Suites Constructed by Different Strategies for *TourApp*

Strategy	Criterion	Sizes of Candidate Test Suites							
		1	2	4	8	16	32	64	128
CARS-H	PU	**13**	**13**	**13–14**	**13–15**	**13–15**	**13–15**	**13–16**	**14–16**
	PUCU	**13**	**13**	**13–14**	**13–15**	**13–15**	**13–15**	**13–16**	**14–16**
	CU	**13**	**13–14**	**13–14**	**13–15**	**13–15**	**13–15**	**13–15**	**13–15**
	CUPU	**13**	**13–14**	**13–14**	**13–15**	**13–15**	**13–15**	**13–15**	**13–15**
	AD	**13**	**13–14**	**13–14**	**13–15**	**13–15**	**13–15**	**13–15**	**14–16**
	AU	**13**	**13**	**13–14**	**13–15**	**13–15**	**13–15**	**13–15**	**13–15**
CARS-L	PU	**6–7**	**4–5**	**1–3**	**1**	**1**	**1**	**1**	**1**
	PUCU	**6–8**	**4–5**	**1–2**	**1**	**1**	**1**	**1**	**1**
	CU	**6–7**	**4–5**	**1–2**	**1**	**1**	**1**	**1**	**1**
	CUPU	**6–7**	**4–5**	**1–2**	**1**	**1**	**1**	**1**	**1**
	AD	**8–9**	**5–6**	**1–4**	**1–2**	**1–2**	**1**	**1**	**1**
	AU	**6**	**3–5**	**1–2**	**1**	**1**	**1**	**1**	**1**
CARS-E	PU	10	10	10	10	10–11	10–11	10–11	10–12
	PUCU	10	10	10–11	10–11	10–11	10–12	10–12	10–12
	CU	10	10	10	10	10–11	10–11	10–11	10–12
	CUPU	10	10	10–11	10–11	10–11	10–11	10–11	10–12
	AD	10–11	10–11	10–11	10–11	10–12	10–12	10–12	10–12
	AU	10	10	10	10	10–11	10–11	10–11	10–12
BS	PU	10	10	10	10	10	10	10	10
	PUCU	10	10	10	10	10	10	10	10
	CU	10	10	10	10	10	10	10	10
	CUPU	10	10	10	10	10	10	10	10
	AD	10–11	10–11	10–11	10–11	10–11	10–11	10–11	10–11
	AU	10	10	10	10	10	10	10	10