To appear in *Proceedings of the 10th International Conference on Quality Software* (*QSIC 2010*), IEEE Computer Society Press, Los Alamitos, CA (2010)

# Correlating Context-Awareness and Mutation Analysis for Pervasive Computing Systems*

Huai Wang, Ke Zhai, and T.H. Tse†
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
{hwang, kzhai, thtse}@cs.hku.hk

***Abstract*—Pervasive computing systems often use middleware as a means to communicate with the changing environment. However, the interactions with the context-aware middleware as well as the interactions among applications sharing the same middleware may introduce faults that are difficult to reveal by existing testing techniques. Our previous work proposed the notion of context diversity as a metric to measure the degree of changes in test inputs for pervasive software. In this paper, we present a case study on how much context diversity for test cases relates to fault-based mutants in pervasive software. Our empirical results show that conventional mutation operators can generate sufficient candidate mutants to support test effectiveness evaluation of pervasive software, and test cases with higher context diversity values tend to have higher mean mutation scores. On the other hand, for test cases sharing the same context diversity, their mutation scores can vary significantly in terms of standard derivations.**

***Keywords—context diversity; mutation analysis; pervasive computing***

## I. INTRODUCTION

Pervasive computing systems provide smart services to users by capturing environment attributes as contexts and adjusting software behaviors according to changes in context values. For example, a smart phone may collect the user's location and activity information to decide the notification mode of incoming calls: it may vibrate silently when the user is in a meeting, but may beep loudly when the user is watching a football match.

The development of pervasive software, however, is challenging. First, the software needs to manage the contexts, including the collection of diverse context information from various sources, cleaning up noisy contexts, classifying the contexts into different categories in an application-usable manner, interpreting low-level contexts, and reasoning about the contexts to provide semantic-level information for the application. Second, the software also needs to communicate with its computing environment dynamically, such as to retrieve changing contexts from data sensing components, record generated contexts to data storage components, and exchange contexts with other components wirelessly. To ease these development obstacles, many context-aware studies [2][4][5][25][35][36] proposed the use of middleware to transparently acquire, disseminate, and interpret the contexts on behalf of the applications over ad hoc networks.

Although such a middleware-centric multi-tier architecture favors the development and configuration of pervasive software, it also brings new challenges to quality assurance. An application may subscribe to its interested contexts and define adaptation rules to specify when to activate adaptive actions. Accordingly, the middleware assembles these subscribed contexts and evaluates adaptation rules to decide whether it should trigger such actions. In this way, the explicit interactions between the application and middleware distribute the application logic over multiple tiers, and hence the faulty states of a program execution may be propagated among multiple architectural levels. Moreover, applications sharing the same context-aware middleware may interact implicitly through their own features, generally known as feature interactions [16]. As a result, the adaptation rules of multiple applications may conflict with one another when manipulating the same stream of context values. In [31], we argued that the changes in context values captured in a test input may play a key role in addressing these testing challenges brought by context-aware properties of pervasive software, and proposed to study context diversity as a metric to measure context changes inherent in test inputs.

In addition, fault-based mutants produced by various mutation operators have been widely used in empirical



† This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 716507 and 717308).

‡ All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2193. Email: thtse@cs.hku.hk.

studies in program testing [3][15][21]. They can accurately simulate the effectiveness of a testing technique over real-life faults [1]. Since no mutation operator for pervasive applications has been proposed in the literature, researchers may use existing mutation operators to produce mutants for pervasive software. Thus, a natural research question would be: *What is the quality of mutants generated by existing mutation operators that are not targeted for pervasive software*?

In this paper, we present a case study that investigated selective properties of faults in multi-tier pervasive software and examined the relationships between context diversity and mutation analysis. We measured the "quality of mutants" from three different dimensions, namely, the number of mutants generated, the ratio of equivalent mutants to all generated mutants, and the ease of killing mutants. We also investigate the trend on the number of mutants killed by individual test cases according to various context diversity values. It helps researchers understand more about the relationship between faults and context-aware inputs when developing verification and validation strategies.

The contribution of this work is two-fold: (1) It confirms that traditional mutation operators can generate a sufficient number of candidate mutants to support the statistical analysis of experiments on pervasive software testing. (2) It reveals that test cases with higher context diversity values tend to have higher mean mutation scores, but for test cases with the same context diversity, their mutation scores can vary significantly in terms of standard derivation values.

The rest of the paper is organized as follows: Section II introduces the preliminaries of our work. Section III states the research questions to be explored. Section IV explains the experimental setup, and Section V presents the findings and analysis results. Section VI reviews related work, followed by a conclusion in Section VII.

## II. PRELIMINARIES

This section recapitulates the background of our case study.

### A. Terminology in Mutation Analysis

To generate faulty versions automatically, one may apply a set of *mutation operators* to the code to produce *mutants*, which are variants of the correct program containing at most one fault. This technique to generate faulty versions is called *mutant generation*. The process to analyze when mutants fail and which test cases trigger such failures is referred to as *mutation analysis* [1].

Formally, given a set of mutation operators $\{\mu_1, \mu_2, \ldots \mu_n\}$, each $\mu_i$ is a function that takes a program $P$ as input and produces a set of mutants $\mu_i(P)$. The program $P$ is referred to as the *golden version*, which is regarded as being free from major faults. The output of a specific execution of a mutant $M$ (or the golden version $P$) on a test case $t$ is denoted by $M(t)$ (or $P(t)$, respectively). Then, a test case $t$ is said to *kill* a mutant $M$ if $M(t) \neq P(t)$ [22] (which means that the mutant $M$ generates an output different from that of the golden version $P$ when $M$ and $P$ take $t$ as input), and $t$ is referred to as *failed test case* with respect to $M$. The *mutation score* of a test case $t$ with respect to a set of mutants $\{M_1, M_2, \ldots, M_n\}$ is defined as the percentage of the mutants killed by $t$.

For each mutant $M$, we define its *perceived failure rate* as $FR(M) = \frac{|\{t \in TP | M(t) \neq P(t)\}|}{|TP|}$, which is the ratio of the number of failed test cases $|\{t \in TP \mid M(t) \neq P(t)\}|$ to the test pool size $|TP|$. For example, a mutant with the perceived failure rate of 0.2 means that 20% of the test cases in the test pool can kill this mutant, and the probability of 10 random test cases to kill it is $1 - (1 - 0.2)^{10} = 99.99\%$. Based on the perceived failure rate, every mutant is classified as an equivalent, must-fail, or normal mutant. A mutant $M$ is said to be an *equivalent mutant* (with respect to the golden version $P$) if its perceived failure rate is 0.0, which means that no test case in the entire test pool can kill $M$. A mutant $M$ is said to be a *must-fail mutant* if its perceived failure rate is 1.0, which means that every test case in the test pool will kill this mutant. Mutants with perceived failure rates within the range of (0.0, 1.0) are referred to be *normal mutants*. Both must-fail mutants and normal mutants are *non-equivalent mutants*.

### B. Terminology of Pervasive Software

Pervasive software often monitors continuously the values of captured contexts. A *context variable* $v$ is a characterization of contexts. We model it as a tuple ($field_1$, $field_2$, …, $field_u$) such that each $field_w$ ($w$ = 1, 2, …, $u$) is an environment attribute used by the pervasive software [18]. A *context instance ins*($v$) is an instantiated context variable such that every field in $v$ has been instantiated. It is a tuple ($f_1$, $f_2$, …, $f_u$) such that each $f_w$ ($w$ = 1, 2, …, $u$) takes the form of ($field_w$ = $value_w$: $type_w$, *timestamp*), where $value_w$, $type_w$, and *timestamp* are the instantiation value, data type, and sampling time for $field_w$, respectively. For ease of presentation, we will write $value_w$ instead of ($field_w$ = $value_w$: $type_w$, *time*). A *context stream cstream*($v$), serving as test inputs of pervasive software, is a time series of the form $\langle ins(v)_{t_1}, ins(v)_{t_2}, \ldots, ins(v)_{t_m} \rangle$, where each $ins(v)_{t_s}$ ($s$ = 1, 2, …, $m$ and $t_s < t_{s+1}$) in *cstream*($v$) is a context instance sampled at time $t_s$.

Let us give an example. Suppose a smart phone 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 a report). As time goes on, the context stream sequence may capture a series of activities, such as (meeting room, present a report), (meeting room, listen to music), and (home, sleep).

### C. Terminology of Context Diversity

*Context diversity* [30] *CD*(*cstream*($v$)) is a metric that measures the amount of context changes inherent in a context stream *cstream*($v$). More precisely, we compute the Hamming distance [13] between every pair of consecutive instances in a given context stream. It is formulated as

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

where $HD(ins(v)_i, ins(v)_{i+1})$ is the Hamming distance [13] between a pair of context instances $ins(v)_i$ and $ins(v)_{i+1}$ in the context stream *cstream*($v$), and $l$ is the length of the context stream *cstream*($v$). Take the context stream introduced in

Section II.B for an example. The sum of Hamming distances for the *location* dimension of the context variable is 1, and that for the *activity* dimension is 2. The context diversity of the sequence is, therefore, 3.

A higher context diversity of a test case can result either from a longer context stream or from more intensive context changes. To facilitate identifying the impact of context changes to the effectiveness of test cases, we partition all test cases into equivalence classes, each class containing test cases with the same context stream length. We refer to each class as a *length-equivalent class* in the rest of the paper.

## III. Research Questions

We present in this section the research questions to investigate in our case study.

*RQ*1: *Do mutation operators for traditional programs generate high-quality mutants for pervasive software?* Since there are no specific mutation operators for pervasive software, it is desirable to explore whether it is favorable to use traditional mutation operators in the setting of pervasive software. By answering this research question, we may be able to identify the applicability of mutation analysis to middleware-based pervasive software.

*RQ*2: *Is there any correlation between context diversity and mutation score of a test case?* In our previous work [30], we argued that context changes contained in individual test cases play a key role in testing context-aware properties of pervasive software, and proposed the concept of context diversity to capture such information. However, we did not explore the kind of context diversity (namely, higher or lower diversity) that can contribute more to the improvement of test effectiveness. More importantly, it is desirable to determine whether test cases with higher context diversity caused by more intensive context changes contribute to better test effectiveness when the context stream length is fixed. By addressing this research question, it can provide guidance to the selection of more efficient test cases.

## IV. Experimental Setup

We describe the set up of the experiment in this section.

### A. Subject

We used *WalkPath* as the middleware-based pervasive software in our empirical study. It was also used as the subject in [18][19]. It consists of a middleware *Cabot* [35] and an application that implements LANDMARC, an RFID-based location sensing algorithm [23]. *Cabot* tracks a person's walking path in indoor space equipped with RFID sensors. The person's current location is obtained via LANDMARC by capturing and analyzing the RFID context. *WalkPath* utilizes the location data as incoming contexts and optionally accepts or repairs them through a set of Context-Inconsistency-Resolution (CIR) services [35] provided by *Cabot*. When a person moves, the application senses their location from the surroundings and reacts accordingly. The whole system is written in Java.

In our case study, *WalkPath* was modeled in three architectural levels, namely the application, middleware, and interface levels. The application level defined the adaptive behaviors, the middleware level managed the context and evaluated the adaptation rules subscribed by the application, and the interface level consisted of a caller, callee, or callback statement to enable interactions between the application and the middleware.

To ensure that the experiment would end within manageable time, we downsized the middleware to contain only critical components (such as context acquisition, context reference, and inconsistency resolution) to support pervasive software capability for the execution of *WalkPath*, and removed other components such as context remote-transmission and database access. The application, the middleware, and the interface contained 231, 483, and 83 non-commented lines of code (LOCs), respectively. Thus, the total size of the adapted *WalkPath* was 797 LOCs.

### B. Mutation Operators and Fault-Based Mutants

To the best of our knowledge, no mutation operator specific to pervasive software has been proposed in the literature. We chose to use the mutation operators proposed for Java programs to produce faulty versions for *WalkPath*. According to [30], there are 12 method-level and 29 class-level mutation operators for Java programs, but one class-level mutation operator (*AMC*) tended to create either equivalent mutants or mutants failing to be compiled, and was excluded from the analysis in [20]. After skipping *AMC*, we used MuClipse [1] to generate fault-based mutants according to all the remaining 40 mutation operators. We used all the generated faults in our study.

We then applied the entire test pool to obtain execution statistics for every mutant. For each mutant, we collected diverse statistics information for further analysis, such as the sets of failed test cases and their context diversity distributions.

### C. Classification Criteria

We classified all the 40 mutation operators according to their respective fault natures [9], namely, missing constructs, wrong constructs, and extraneous constructs. A fault is categorized as a missing construct if a variable assignment or initialization, logical condition, parameter or expression in a function call, or part of an algorithm or program module is missing or incomplete. Faults due to wrong or extraneous constructs are similarly defined. The classification results are shown in Table I. We observe that, out of the 40 mutation operators, 12 are used to delete program constructs, 18 are applied to replace or modify program constructs, and the remaining 10 are employed to insert program constructs.

After partitioning the mutation operators, we examined the locations of the mutants they produced. We labeled each mutant according to one of the three architectural levels. This enabled us to study the distributions of the mutants over the architectural tiers. The results will be discussed in Section V.A.

[1] Available at http://muclipse.sourceforge.net.

TABLE I. CLASSIFICATION OF MUTATION OPERATORS BY FAULT NATURE.

| Fault Nature | Mutation Operators | Count |
|---|---|---|
| Missing Construct | AOD, COD, LOD, IHD, IOD, ISD, IPC, PCD, OMD, JTD, JSD, JID | 12 |
| Wrong Construct | AOR, ROR, COR, SOR. LOR, ASR, IOP, IOR, PMD, PPD, PCC, PRV, OMR, OAC, EOA, EOC, EAM, EMM | 18 |
| Extraneous Construct | AOI, COI, LOI, IHI, ISI, PNC, PCI, JTI, JSI, JDC | 10 |
| **Total** | | 40 |

### D. Test Pool and Context Streams

In our experiment, we reused an existing test pool for *WalkPath*, which contained 20,000 different test cases, each consisting of real-world data captured via RFID readers and used in previous experiments [18][19]. The test pool was shown to be capable of generating test sets that fulfill different data-flow testing criteria [18][19].

A test case for *WalkPath* consists of a sequence of locations as inputs. In *WalkPath*, *Cabot* tracks a user's location and reports the estimated coordinates to the application. After receiving the coordinates, the application invokes CIR services of *Cabot* to detect potentially inconsistent contexts. After the user has ended the walk, the application reports the trace of the user's locations. In our experiment, we used the sequence of location data as the context stream for *WalkPath*.

### E. Test Effectiveness Measure

The effectiveness of a test case $c$ is simulated by its mutation score [24]. More formally, given a mutation operator $\mu_i$ and a program $P$, $Nm_i(P)$ represents the number of normal mutants generated by applying $\mu_i$ to $P$, and $Km_i(P)$ denotes the number of normal mutants from $Nm_i(P)$ killed by $c$. The test effectiveness of $c$ with respect to mutants generated by the mutation operator $\mu_i$ is measured by $\frac{Km_i(P)}{Nm_i(P)}$, and the test effectiveness of $c$ with respect to a set of mutants $\{Nm_1(P), Nm_2(P), \ldots, Nm_n(P)\}$ is measured by $\frac{\sum_{i=1}^{n} Km_i(P)}{\sum_{i=1}^{n} Nm_i(P)}$. Our metric to measure the effectiveness of a test case is similar to that in other work such as [24].

## V. DATA ANALYSIS OF EMPIRICAL STUDY

We present our empirical findings in this section.

### A. Answering *RQ*1: *Quality of Generated Mutants*

We categorize all the generated mutants according to the architectural levels and fault natures and show the results in Table II. We observe that, MuClipse does generate a large number of mutants (4884) for pervasive software. On the other hand, the total number of mutants labeled with missing constructs is very small (only 11 out of 4884). This result contradicts the presence of widely-observed omission faults in real-world practices [9][33] as well as the experiments results in [20], which reports that the mutation operators *IOD*, *JID*, and *JTD* can generate 496, 115, and 203 missing construct mutants when applied to 264 BCEL classes[2]. After examining our code in detail, we found that this inconsistency may be caused by the following factors: (1) The inheritance features for object-oriented software are not widely-used in the adapted *WalkPath*, which makes it hard for *IOD* to find overriding methods to delete. (2) Many member variables of classes are modified by the keyword "final" in the adapted *WalkPath*, which makes it hard for *JID* to delete initializations of member variables. (3) The keyword "this" is omitted as long as the omission does not cause compilation errors in the currently adapted *WalkPath*, which makes it hard for *JTD* to find the keyword "this" to delete.

TABLE II. CLASSIFICATION OF MUTANTS BY ARCHITECTURAL LEVEL AND FAULT NATURE.

| Architectural Level \ Fault Nature | Missing Construct | Wrong Construct | Extraneous Construct | Total |
|---|---|---|---|---|
| **Application** | 3 (0.06%) | 1490 (30.51%) | 2438 (49.92%) | 3931 (80.49%) |
| **Middleware** | 3 (0.06%) | 379 (7.76%) | 409 (8.37%) | 791 (16.20%) |
| **Interface** | 5 (0.10%) | 81 (1.66%) | 76 (1.56%) | 162 (3.32%) |
| **Total** | 11 (0.23%) | 1950 (39.93%) | 2923 (59.85%) | 4884 (100%) |

Neither equivalent mutants nor must-fail mutants are good candidates for test effectiveness evaluation because they may underestimate and overestimate the effectiveness of testing techniques, respectively. Thus, a desirable mutant tool should generate a large number of (normal) mutants that are neither equivalent nor must-fail.

TABLE III. EQUIVALENT AND NON-EQUIVALENT MUTANTS GENERATED BY MUCLIPSE.

| Architectural Level | Equivalent Mutants | Non-equivalent Mutants | | Total |
|---|---|---|---|---|
| | | Must-fail Mutants | Normal Mutants | |
| **Application** | 1946 (39.84%) | 592 (12.12%) | 1393 (28.52%) | 3931 (80.49%) |
| **Middleware** | 277 (5.67%) | 14 (0.29%) | 500 (10.24%) | 791 (16.20%) |
| **Interface** | 57 (1.17%) | 64 (1.31%) | 41 (0.84%) | 162 (3.32%) |
| **Total** | 2280 (46.68%) | 670 (13.72%) | 1934 (39.60%) | 4884 (100%) |

We present the number of equivalent, must-fail and normal mutants generated by MuClipse in Table III. It shows that 46.68% (2280 out of 4884) of the mutants produced by MuClipse are equivalent mutants. The ratio of equivalent mutants in Dahm [7] was lower than ours. They reported that the ratio of equivalent mutants is 31.59% (187 out of 592) when applying the same set of mutation opera-

[2] Available at http://jakarta.apache.org/bcel/

tors to traditional Java programs. As mentioned in Section II.A, we only considered the final outputs of the golden version and the mutants to identify the equivalent mutants. On the other hand, Dahm [7] used a different oracle (weak mutation [14]) to determine equivalent mutations. The inconsistent results suggest that it is necessary to further investigate the equivalent mutant problem by varying the strength of the oracle used to kill mutants. In addition, the higher equivalent mutant ratios (in relation to traditional software) may also confirm the intuition that pervasive software can be more observable than traditional software because it generates contexts to the external computation environment.

Second, 13.72% (670 out of 4884) generated mutants were must-fail mutants. The two above-mentioned statistics showed that 60.40% of the mutants were unsuitable for test effectiveness evaluation owing to their extreme perceived failure rates (namely, a low perceived failure rate of 0.0 for equivalent mutants and a high perceived failure rate of 1.0 for must-fail mutants).

Owing to page constraint, we only present the detailed statistics of faults based on the classification of their architecture levels in Figure 1 and Figure 2, and skip the corresponding results based on the classification of fault natures. Figure 1 partitions all the 1934 (39.6%) normal mutants into three architectural levels, and compares their results with the overall result, which we refer to as "all levels". The figure shows that, for the all-levels category (that is, the leftmost bar), the minimum, median, mean, maximum, and standard deviation of perceived failure rates are 0.00005, 0.11428, 0.23235, 0.784350, and 0.25267, respectively. The corresponding values for the application level are 0.00005, 0.11925, 0.26289, 0.784350, and 0.26896, respectively. For the middleware level, they are 0.00005, 0.08515, 0.13979, 0.77265, and 0.16726, respectively, and for the interface level, they are 0.01980, 0.20895, 0.32358, 0.77410, and 0.26623, respectively. In terms of the mean perceived failure rates, MuClipse tends to generate mutants with high perceived failure rates at all four levels. Take a mutant with the mean perceived failure rate of 0.13979 at the middleware level as an example. It only needs 20 test cases to kill this mutant with a probability of $1 - (1 - 0.13979)^{20} = 95\%$.

Figure 2 groups these 1934 normal mutants at various architectural levels based on their perceived failure rates and provides us with more detailed analysis of the characteristics of the generated mutants. Figure 2(a) shows that, for all normal mutants in the application level, 41.0% (571 out of 1393) have perceived failure rates of no more than 0.1, which means that a test suite with 30 test cases has a probability of $1 - (1 - 0.1)^{30} = 95.77\%$ to kill 59.0% (822 out of 1393) of the mutants. When manifesting the range (0.0, 0.1] for these 571 mutants, we find that 3.3% (46 out of 1393) of the mutants having a perceived failure rate within the range of (0.00, 0.01], and 4.8% (67 out of 1393) of the normal mutants within the perceived failure rate range of (0.05, 0.06]. If we regard perceived failure rates no more than 0.06 as a cut off, then application-, middleware-, and interface-level mutants can contribute 23.12% (322 out of 1393, with a mean perceived failure rate and a standard derivation of 0.033 and 0.0179, respectively), 44.80% (224 out of 500, with a mean perceived failure rate and a standard derivation of 0.028 and 0.0184, respectively), and 4.88% (2 out of 41, with a mean perceived failure rate and a standard derivation of 0.020 and 0.0000, respectively) of the fault candidates. In other words, if we assume that 0.06 is the upper bound for perceived failure rates of mutants in test effectiveness evaluation, then MuClipse can provide 28.34% (548) of the fault candidates among 1934 normal mutants. The ratio is not high, but the number of normal mutants whose perceived failure rates are within the range of (0.00, 0.06] can be large enough for test effectiveness evaluation.

In summary, in order to answer research question *RQ*1, we measured "the quality of mutants" generated by traditional mutation operators for pervasive software from the number of mutants generated, the ratio of equivalent mutants, and the ease of killing mutants. The results show that traditional mutation operators can generate a large number (4884) of compilable mutants for pervasive software, among which 46.68% (2280 out of 4884) are equivalent mutants and 11.22% (548 out of 4884) have perceived failure rates within the range of (0.00, 0.06] that can be fault candidates used in test effectiveness evaluation. The middleware-level can contribute most to fault candidates in terms of fault percentage (44.80%), while the application-level can generate the largest number (322) of fault candidates. In other words, traditional mutation operators can generate sufficient candidate mutants to support statistical analysis for experiments in pervasive software testing. In addition, our results also suggest that the specific setting of the adapted *WalkPath* may cause traditional mutation operators to fail to simulate widely-observed omission faults in practice, and our oracle used in killing mutants may also contribute to the relatively high proportion of equivalent mutants. Therefore, further experimental studies involving in more subjects and different oracles will be required to address the limitations related to this case study.

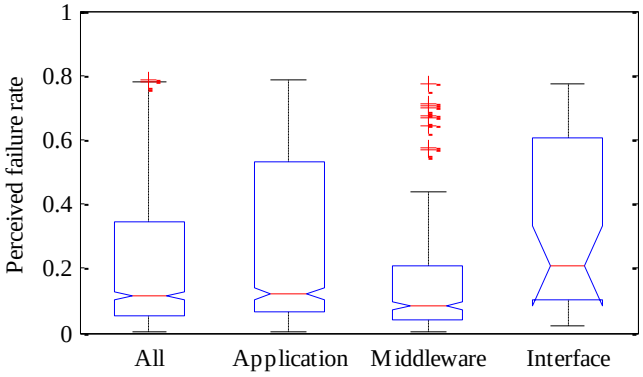

Figure 1. Perceived failure rates of mutants in various architectural levels.

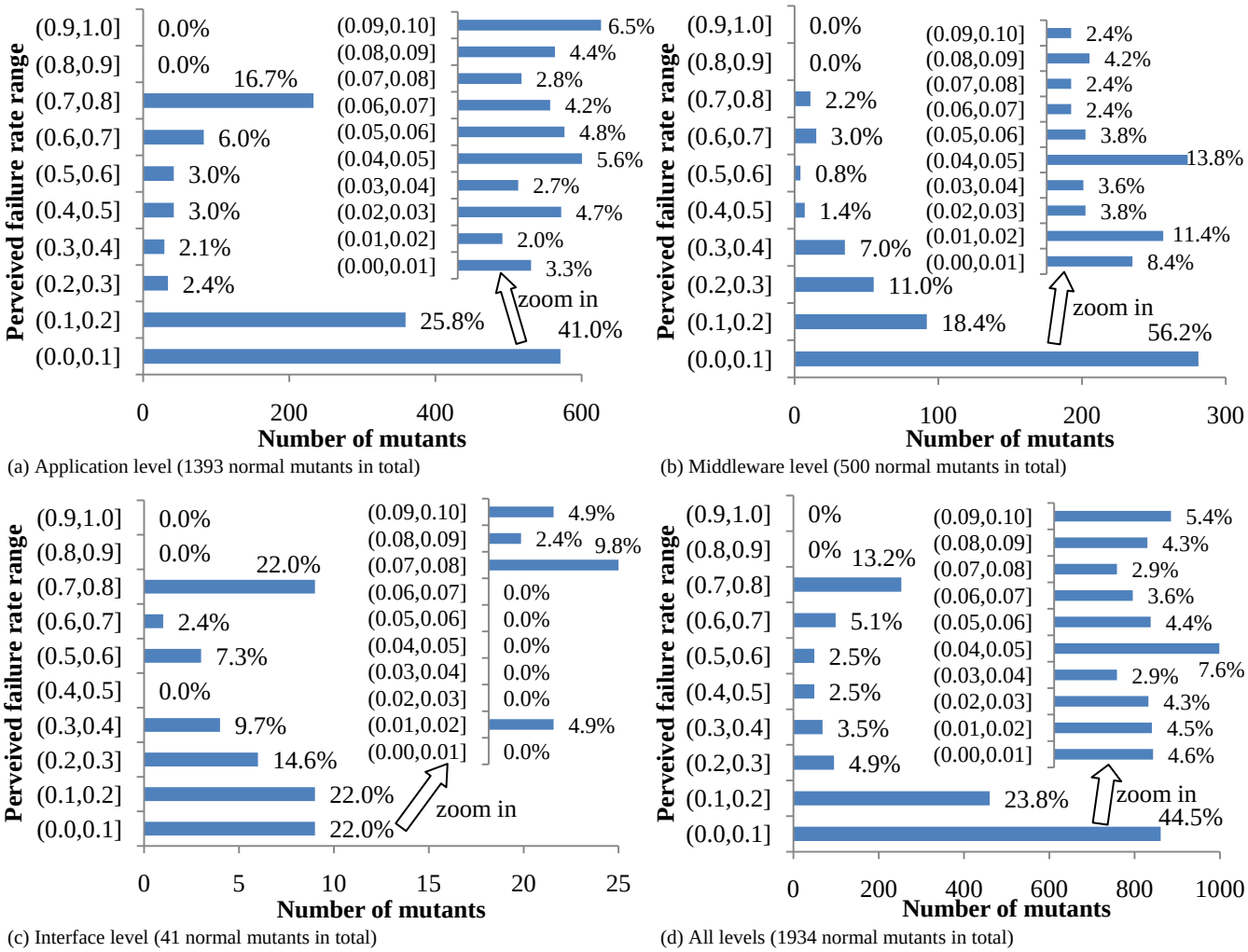


Figure 2. Perceived failure rates of normal mutants in various architectural levels for *WalkPath*.

### B. *Answering RQ2: Correlation between Context Diversity and Mutation Score*

To study the correlation between the context diversity and mutation score of a test case, we partitioned the set of all mutants according to the architectural level and fault nature. We divided the mutants into the application, middleware, and interface levels. For the purpose of comparison, we also considered all the mutants as one group, denoted by the "all" level. Similarly, we classified mutants into missing constructs, extraneous constructs, wrong constructs, and all constructs. Under each architectural level or fault nature, we grouped all the test cases sharing the same context diversity value into the same set, and used the corresponding context diversity value as the identifier of this set. For each such set, we computed the mean mutation score for all the test cases in the set. The results for various architectural levels and fault natures are shown in Figure 3 and Figure 4, respectively.

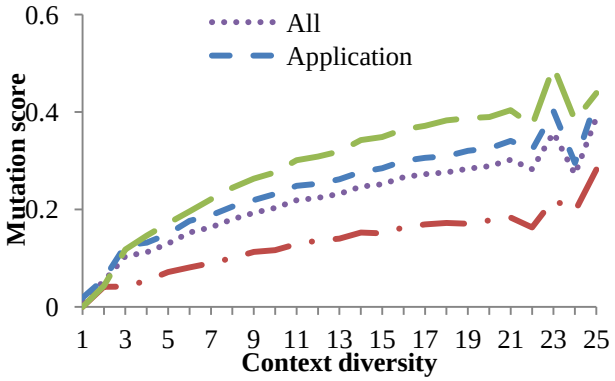


Figure 3. Mutation scores of test cases for mutants in various architectural levels.

We observe from Figure 3 that, at each architectural level, the mean mutation scores of test cases increase when the test cases have higher context diversity. This indicates that there is a positive correlation between the context diver-

sity and mutation score of a test case at each level. To verify whether such a positive correlation indeed exists, we further conducted the Pearson correlation test. As shown in Table IV, all Pearson correlation coefficients are larger than 0.9. This confirms that, for mutants in every architectural level, there is a strong correlation between the context diversity and mutation score of a test case. Furthermore, from Figure 3, there is a large difference of mean mutation scores between two levels. For instance, for test cases attaining a context diversity of 13, the differences in mean mutation scores between the application level and the interface level can be more than 20%.

TABLE IV. PEARSON CORRELATION COEFFICIENTS (PCC) AT VARIOUS ARCHITECTURAL LEVELS.

| | All Levels | Application Level | Middleware Level | Interface Level |
|---|---|---|---|---|
| **PCC** | 0.9527 | 0.9472 | 0.9573 | 0.9400 |

We also observe from Figure 4 that, the mutation score of a test case generally increases as the context diversity of the test case increases. We also conducted a Pearson correlation test and the results are shown in Table V. All Pearson correlation coefficients in the table are also larger than 0.9. This further confirms that, for mutants with any of the four fault natures, there is a positive correlation between the context diversity and mutation score of a test case. Again, the differences in mutation scores among fault natures are observably significant.

TABLE V. PEARSON CORRELATION COEFFICIENTS (PCC) FOR VARIOUS FAULT NATURES.

| | All Constructs | Missing Construct | Wrong Construct | Extraneous Construct |
|---|---|---|---|---|
| **PCC** | 0.9527 | 0.9409 | 0.9615 | 0.9637 |

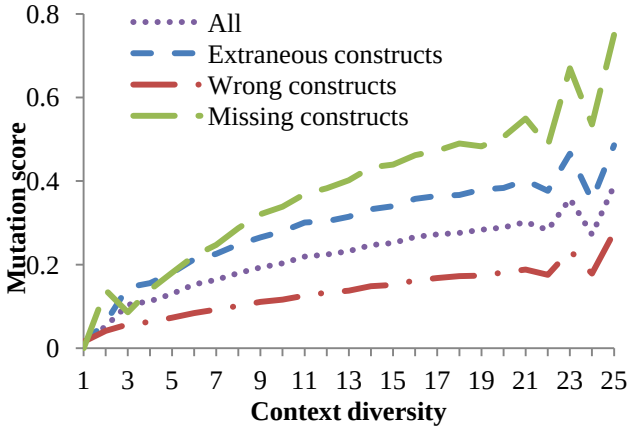


Figure 4. Mutation scores of test cases on mutants with various fault natures.

It is worth noting that, although test cases with higher context diversity values tend to have higher mean mutation scores, for test cases sharing the same context diversity values, their mutation scores vary significantly in terms of standard derivations. (We omit the detailed data owing to space constraints.). Further dynamic analysis will be required to find out how a testing method may apply context diversity effectively in practice.

One may wonder whether a longer context stream or more intensive context changes may contribute more to mutation scores of test cases. Thus, it is natural to ask how the amount of context changes correlates with mutation scores when the context stream length is fixed. To investigate this problem, we further partitioned all the test cases into length-equivalent classes (see Section II.C for the corresponding definition). Owing to the page limit and the large amount of data, we only present the all-levels category.

Out of the 24 length-equivalent classes, 12 sets with context stream lengths $11-14$, $19-22$, and $23-26$ give similar results to either the set with length $7-10$ or the set with length $15-18$. For ease of presentation, we only show 12 representative sets with context stream lengths $7-10$, $15-18$, and $27-30$ in Figure 5. For each plot in the figure, the $x$-axis represents the context diversity while the $y$-axis shows the mean mutation score of the test cases having the same context diversity length. To extract the implicit correlation between the context diversity and mutation score of a test case, we used a linear model $y = ax + b$ to fit the raw data, in which $x$ and $y$ were substituted by the context diversity and mutation score, respectively, and $a$ and $b$ were parameters to be solved by the model. We show 12 fitted lines (each representing one length-equivalent class) in Figure 5 and their corresponding fitness parameters in Table VI.

We observe that every line in Figure 5 has a positive slope. Take the line labeled with "length = 7" as an example. It shows that the average mutation score increases from 0.288 to 0.598 when the context diversity of test cases increases from 10 to 20. This observation is also confirmed by the data from Table VI, where all the values in the column "$a$" representing the slopes of corresponding lines are larger than 0. Since all the test cases on the line have the same context stream length, the only identified independent variable that can contribute to the increase in context diversity of these test cases is the context change. That is, the amount of context changes does have positive correlations with the effectiveness of test cases in terms of mean mutation scores.

Furthermore, we observe from Figure 5 that, in most cases (except the lines labeled with "length = 9", "length =28", and "length = 29"), the slopes of the lines decrease with context stream length. For instance, when comparing between the lines labeled with "length = 7" and "length = 8" in Figure 5, we find that the former line is steeper than the latter. This can be further confirmed by the data from Table VI, where the value of "$a$" decreases from 0.031 to 0.020 when the length increases from 7 to 8, and this trend holds except the data with lengths 9, 28, and 29. Both observations may suggest that the positive correlation between the context changes and mutation scores of test cases become weaker with the increase of the context stream length.

TABLE VI. CURVE-FITTING PARAMETERS FOR DATA IN FIGURE 5

| Length | *a* | *b* | Sum of Square of Errors |
|---|---|---|---|
| **7** | 0.031 | –0.022 | 0.000 |
| **8** | 0.020 | 0.020 | 0.011 |
| **9** | 0.027 | –0.027 | 0.003 |
| **10** | 0.018 | 0.042 | 0.004 |
| **15** | 0.010 | 0.106 | 0.003 |
| **16** | 0.011 | 0.086 | 0.001 |
| **17** | 0.011 | 0.099 | 0.001 |
| **18** | 0.007 | 0.150 | 0.011 |
| **27** | 0.005 | 0.193 | 0.004 |
| **28** | 0.017 | –0.024 | 0.002 |
| **29** | 0.009 | 0.085 | 0.029 |
| **30** | 0.002 | 0.294 | 0.027 |

On the other hand, we observe from Figure 5 a mixed result for the correlation between the context stream length and mutation score of test cases with the same context diversity. Some pairs of lines show a positive correlation between the context stream lengths and mutation scores of test cases when their context diversity is fixed. For example, given any specific context diversity, test cases on the line labeled with "length = 27" always achieve lower mutation scores than those on the line labeled with "length = 30". However, counterexamples exist to falsify such positive correlation for a fixed context diversity. For instance, given a fixed context diversity larger than 4, test cases on the line labeled with "length = 7" always obtain higher mutation scores than those on the line labeled with "length = 10". This observation may imply that, after a given context diversity has been obtained, further increase of the context stream length does not necessarily improve the mutation scores of test cases.

(*) If the lengths of the context streams are fixed, the context diversity may have a positive correlation with the mutation scores of test cases. However, as the lengths of context streams increase, the correlation between context changes and mutation scores may become weaker, even though it still remains positive. In addition, given a context diversity value, increasing the context stream length does not necessarily improve the mutation scores of test cases.

To confirm the significance of the correlation, we conducted Pearson correlation tests on all groups of data. We found that there was a mild to strong positive correlation between the context changes and mutation scores of test cases. This empirical result may further help researchers develop new verification and validation techniques.

In summary, in order to answer research question *RQ*2, we have studied the correlation between the context diversity and mutation scores of a test case. We have at least three observations based on the empirical results. First, there is a strong positive correlation between the context diversity and mutation scores of test cases, which implies that test cases with higher context diversity values seem to present higher mean mutation scores. On the other hand, another observation that the large variances for the mutation scores of test cases sharing the same context diversity also suggests that such an improvement in test effectiveness is not stable. Second, the correlation between context diversity and mutation scores holds if the context stream length is fixed, and as the context stream lengths increase, the correlation becomes weaker but still remains positive. Third, after a given context diversity has been obtained, further increase of the context stream length does not necessarily result in higher mutation scores.

### C. Threats to Validity

*Threats to construct validity*. Construct validity relates to whether our defined metrics really measure the properties we intend to capture. We applied the most commonly used metric, mutation score, as a measure for test effectiveness. Other metrics such as the time needed to generate a test suite killing all mutants may produce different results. We used 40 mutation operators for Java programs to generate a large variety of mutants for *WalkPath*. Other mutation operators for different programming languages may produce different results. We measured the quality of generated mutants in terms of the number of generated mutants, the ratio of equivalent mutants to all generated mutants, and the ease of killing mutants. The use of other metrics to measure the quality of mutants may result in a different conclusion for *RQ*1. We have used a linear model $y = ax + b$ to fit the raw data for studying the correlation between context changes and mutation scores of test cases. Other nonlinear regression models such as the exponential model $y = ax^b + c$ in [1][17] or the logistic model $y = \frac{e^x}{1+e^x}$ in [11] may produce different results for *RQ*2.

*Threats to internal validity*. Internal validity refers to the possibility that uncontrolled factors other than our defined metrics (including the mutation operators in *RQ*1 and the context diversity in *RQ*2) are responsible for the results. In our experiment, we have used an existing test pool that contains 20,000 RFID data collected in real-life settings for non-testing experiments, and this test pool has been shown to be large enough in terms of constructing adequate test suites that obtain high coverage with respect to data-flow-based testing criteria [18][19]. The use of different test pools may give different empirical results. To reduce human errors, we implemented a tool to collect the statistics about the context diversity and mutation scores of test cases, as well as the perceived failure rates of mutants. We verified the tool against small programs and spot-checked the results of larger programs.

*Threats to external validity*. External validity is concerned with the extent that we can generalize our empirical results to other subject programs. A major threat of the experiment is probably that we used only one subject *WalkPath* in this case study (although it has been studied extensively in [18][19]). The middleware-based programming model for *WalkPath* is representative for pervasive software, and we have explained the necessity of such layered system architecture in Section I. On the other hand, the results for *RQ*1 would probably vary according to the specific

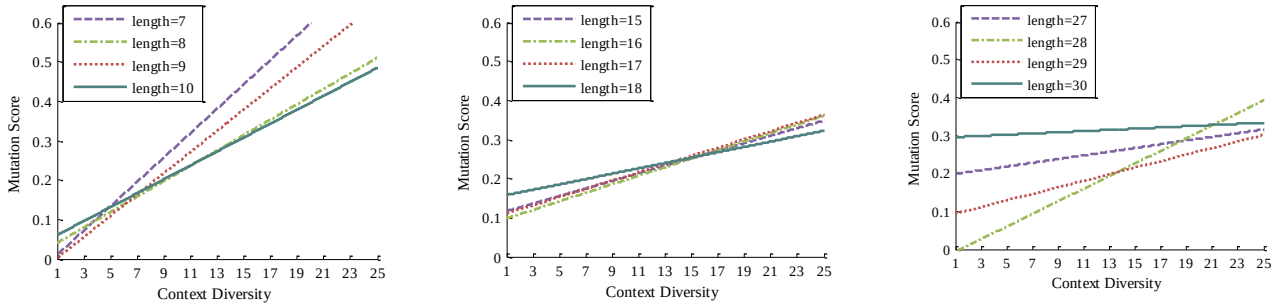


Figure 5. Correlation between context changes and mutation scores.

development process, especially because the natures of the faults may vary at different verification phases. For instance, specific settings of *WalkPath* (such as infrequently-used inheritance features, widely-used field modifier "final", the omission of keyword "this", and so on ) may make the mutation operators fail to generate sufficient missing construct mutants. It will be very important, therefore, to replicate the study on other subject programs using other mutation operators and oracles in order to yield more generalizable results for *RQ*1.

## VI. Related Work

Two seminal papers on mutation testing, which used mutants to measure the adequacy of test suites, are Hamlet [12] and DeMillo et al. [8]. A premise of mutation testing is that test cases which detect simple faults can also detect a large percentage of complex faults composed from simple faults. Offutt et al. [24] have provided empirical support for this important premise. To define and generate simple faults systemically, many different sets of mutation operators and corresponding mutation tools have been proposed for different programming languages, such as Proteum [10] for C programs and MuJava [20] and MuClipse [29] for Java programs. Although empirical results in [1] have shown that the effectiveness of test suites in killing mutants can accurately measure the effectiveness of test suites in finding real faults, it has been reported that the number of mutants generated by mutation operators are usually too large. To solve this practical difficulty in applying mutant testing, various ways of speeding up mutant testing have been proposed, including equivalent mutant elimination [28] and finding a sufficient set of mutation operators [22]. On the other hand, we have conducted mutant analysis in the pervasive software setting and found that traditional mutation operators that are not specific to pervasive applications can still support the testing of pervasive software in generating sufficient mutants.

Our work is also related with the quality assurance of pervasive software. Noting that the output of a test case for pervasive applications can be too transient to record, Chan et al. [6] advocated the use of metamorphic relations among different contexts to address the test oracle problem. Observing that layered architecture disseminated the complete application logic of pervasive software into multiple tiers, Lu et al. [18][19] proposed to assemble context-aware entities (including adaptation rules and context management components such as CIR services) into a traditional control-flow-graph model and developed new coverage-based testing criteria to dynamically verify the definition and use of variables in pervasive software. Wang et al. [34] developed another set of coverage-based testing techniques for concurrent pervasive software. Lai et al. [17] also proposed a set of coverage-based testing strategies to reveal synchronization faults when nesC programs readjust its behaviors to the new context. Roman et al. [26] proposed Mobile UNITY as a model to represent mobile applications and verify them against the specified properties. Sama et al. [27] further developed fault models for context-aware applications. None of these techniques took advantage of context diversity inherent in individual test cases for pervasive software. Our work, therefore, complements these techniques.

## VII. Conclusion

In this paper, we have presented two research questions to investigate the applicability of applying traditional mutation operators to generate mutants for pervasive software, and the correlation between the context diversity and mutation score of a test case. Our empirical results confirm that these mutation operators can generate sufficient numbers of candidate mutants (548 out of 4884 with perceived failure rates within the range of (0.00, 0.06]) to support the statistical analysis of pervasive software testing experiments. We have also found that test cases with higher context diversity tend to have higher mean mutation scores. On the other hand, for test cases with the same context diversity values, the mutation scores can vary significantly. The positive correlation between context diversity and mean mutation scores holds if the context stream length is fixed. However, as the context stream length increases, the correlation becomes weaker, even though it still remains positive. Moreover, we have observed that an increase in the context stream length does not necessarily result in higher mutation scores of test cases after a given context diversity has been obtained. These findings suggest that, in order to improve test effectiveness, it would be a good idea to select test cases with higher context diversity (when different test cases have different context stream lengths) or more intensive context changes (when different test cases have the same context stream length). We have obtained preliminary results in [32] and will report them in more details soon. As future work, we will extend our empirical study to include

more subjects, different mutation operators, and diverse oracles to kill mutants. We will also study how to refine context diversity so that it will more stably contribute to the mutation scores of test cases.

## REFERENCES


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

[2] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless Internet. *IEEE Transactions on Software Engineering*, 29 (12): 1086–1099, 2003.

[3] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering* (*ICSE 2004*), pages 86–95. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[4] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29 (10): 929–944, 2003.

[5] A. T. S. Chan and S.-N. Chuang. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29 (12): 1072–1085, 2003.

[6] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16 (5): 677–703, 2006.

[7] M. Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut fuer Informatik, Freie Universitaet Berlin, Berlin, Germany, 1999.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11 (4): 34–41, 1978.

[9] J. A. Duraes and H. S. Madeira. Emulation of software faults: a field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32 (11): 849–867, 2006.

[10] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Proteum/FSM: a tool to support finite state machine validation based on mutation testing. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society* (*SCCC 1999*), pages 96–104. IEEE Computer Society Press, Los Alamitos, CA, 1999.

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

[12] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3 (4): 279–290, 1977.

[13] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29 (1): 147–160, 1950.

[14] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9 (4): 233–262, 1999.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*), pages 191–200. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[16] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: a survey. *IEEE Transactions on Software Engineering*, 24 (10): 779–796, 1998.

[17] Z. Lai, S. C. Cheung, and W. K. Chan. 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 2008/FSE-16*), pages 94–104. ACM Press, New York, NY, 2008.

[18] H. Lu, W. K. Chan, and T. H. Tse. 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 2006/FSE-14*), pages 242–252. ACM Press, New York, NY, 2006.

[19] H. Lu, W. K. Chan, and T. H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 61–70. ACM Press, New York, NY, 2008.

[20] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15 (2): 97–133, 2005.

[21] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing?. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 164–173. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[22] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 351–360. ACM Press, New York, NY, 2008.

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

[24] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1 (1): 5–20, 1992.

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

[26] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1 (4): 74–83, 2002.

[27] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang. Context-aware adaptive applications: fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 2010. doi: 10.1109/TSE.2010.35.

[28] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the 2009 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2009*), pages 69–80. ACM Press, New York, NY, 2009.

[29] B. H. Smith and L. Williams. An empirical evaluation of the MuJava mutation operators. In *Proceedings of the Testing: Academic and Industrial Conference: Practice And Research Techniques* (*TAICPART-MUTATION 2007*), pages 193–202. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[30] M. Umar. An Evaluation of Mutation Operators for Equivalent Mutants. Project report, MSc in Advanced Software Engineering, Department of Computer Science, King's College London, London, UK., 2006.

[31] H. Wang and W. K. Chan. Weaving context sensitivity into test suite construction. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*

(*ASE 2009*), pages 610–614. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[32] H. Wang, W. K. Chan, and T. H. Tse. On the construction of context-aware test suites. Technical Report TR-2010-01. Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong, 2010.

[33] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 45–55. IEEE Computer Society Press, Los Alamitos, CA, 2009.

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

[35] C. Xu, S. C. Cheung, W. K. Chan, and C. Y. Ye. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology*, 19 (3): Article No. 9, 2010.

[36] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. IEEE Pervasive Computing, 1 (3): 33–40, 2002.