To appear in *The 2009 IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2009*)

Weaving Context Sensitivity into Test Suite Construction†

Huai Wang
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
hwang@cs.hku.hk

W.K. Chan‡
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

***Abstract*—**Context-aware applications capture environmental changes as contexts and self-adapt their behaviors dynamically. Existing testing researches have not explored context evolutions or their patterns inherent to individual test cases when constructing test suites. We propose the notation of context diversity as a metric to measure how many changes in contextual values of individual test cases. In this paper, we discuss how this notion can be incorporated in a test case generation process by pairing it with coverage-based test data selection criteria.

Keywords: *context diversity*; *software testing; context-aware program*

I. INTRODUCTION

Context-aware applications capture the environmental evolutions as *contexts*, and *self-adapt* their behaviors dynamically and contextually. The massive volume of contexts provides unprecedented levels of details about the physical and computational environments surrounding the context-aware applications. Moreover, they should make sequences of contextual and fine-gained decisions and implement such sequences during program executions. It is challenging, however. For instance, there are diverse kinds of noise existed in the captured contexts [13] and significant errors in recovering actual situations based on given contexts even though current state-of-the-art techniques have been used [14]. Still, programmers are expected to develop reliable solutions for the actual situations. Testing is a method to check whether a program behaves as expected and to reveal the presence of faults through executions of test cases. It can be conducted by setting up certain actual situations and observing the corresponding outputs from the program under test.

Many state-of-the-art testing techniques have been proposed to exercise various context-dependent data-flow [5][6] and control-flow [4][5] entities, manipulate the thread interleaving choices at the context-related function call level [12] or at the scheduler level [4], or check the context relationships among executions as alternate test oracles [10]. Experimental studies of these coverage-based testing techniques show that there is still plenty of room for improvement. For instance, in [5], the previous proposed techniques have reported that they are not effective enough to reveal the presence of faults in their "hard" category.

In this paper, we propose to augment coverage-based testing techniques with a supplementary criterion. It is based on a measurement metric, which we call *context diversity*. As we have mentioned above, during program executions, a context-aware program may make a sequence of decisions to adapt its behaviors. We argue that such decisions are often contextual in nature, and a primary source of such information is originated from contexts, or their patterns and evolutions. If it is the case, the sequence of contexts used by a context-aware program (or its underlying middleware) to activate such self-adaptation decisions may be valuable to assure the program quality. On the other hand, even though all such decisions are executed by the program sequentially or in parallel as the expected version of the program does, the potential presence of omission faults in the program may make such sequences differ from how the sequences of contexts to be used by the expected version.

One instance of the supplementary criterion aims to diversify the contextual values in such sequences of contexts. It assumes that self-adaptation occurs when there is a significant change in contextual values. Different projections (in view of the potential presence of omission faults) of a sequence with relatively high consecutive changes in contextual values may associate with different adaptation sequences. The instance measures how many consecutive changes in contextual values in such a sequence. In other instances, we may use the metric to minimize the diversifications.

To use the criterion, one may pair it up with coverage-based techniques. For instance, when a candidate test case is generated by a test suite construction procedure, it may be deemed as redundant if the candidate fails to increase the coverage achieved by the constructing test suite. In this case, our criterion can be used to select a set of test cases, including the candidate and the ones in the constructing test suite to remove. In this removal process, we may use the measurement metric to guide the above-mentioned construction process to improve, worsen, or randomize the context change levels exhibited in the constructing test suite. In this paper, we illustrate such integration.

The main contribution of the paper is threefold: (i) It points out the role of contexts in test suites as a way to

† This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111107 and 123207).

‡ All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Tel: (+852) 2788 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

enhance other test case generation techniques. In particular, it shows the use of information *in* test cases (rather than information *in* programs or its specification) to construct test suites. (ii) It formulates the notion of context diversity to measure the contextual changes in test cases. (iii) It reports an experiment to show how much context diversity can be injected in coverage-adequate test suites.

The rest of this paper is organized as follows: Section II gives the preliminaries of this work. Section III uses a motivating example to illustrate the inadequacy of testing techniques that are neutral to contextual changes in test cases. Section IV presents our idea, and Section V discusses some properties of our technique. Section VI reviews related work, followed by a conclusion in Section VII.

II. PRELIMINARY

In this section, we state the preliminaries to ease the presentation of our technique. A context-aware application *p* may associate with a set of environment attributes, which can be characterized by *context variable* [6]:

Definition 1 (Context Variable): A **context variable** *C* is a tuple ($field_1$, $field_2$, … , $field_n$), where each $field_i$ ($0 < i < n + 1$) represents an environment attribute subscribed by *p*.

However, context variables are symbolic in nature. They quantify the (environmental) patterns that *p* reacts to. To use these contexts in actual program executions, the required context variables should be initialized. A context instance *ins*(*C*) is said to be generated when all fields in a context variable *C* have been instantiated [5][6]:

Definition 2 (Context Instance): A context instance *ins*(*C*) of a context variable *C* is a tuple (i_1, i_2, … , i_n) such that each i_t ($0 < t < n + 1$) takes the form of ($field_t$ =*value*: *type*, *time*), where *value*, *type* and *time* are the instantiation value, data type and sampling time for $field_t$, respectively.

We are interested in how a testing technique uses the contextual value, and thus, in the rest of the paper, we simply write *value* rather than ($field_t$ =*value*: *type*, *time*). A sequence of context instances is called a *context stream*. Typically, the data volume of such context streams is large, and thus, it is generally intractable to test the entire data streams for assuring the program quality. For testing purpose, a test case may therefore only take a fragment of a context stream as a part of its test input.

Definition 3 (Context Stream Fragment): A context stream fragment *cstream*(*C*) of a context variable *C* is an ordered sequence of *ins*(*C*) as $\langle ins(C)_{t_1}, ins(C)_{t_2}, \dots, ins(C)_{t_n} \rangle$. Each $ins(C)_{t_i}$ ($0 < i < n$ and $t_i < t_{i+1}$) in *cstream*(*C*) is a context instance sampled at time t_i and all context instances in *cstream*(*C*) are ordered by t_i.

III. THE PROBLEM

A. Motivating example: Conveyor Belt

We use the application scenario of *Conveyor Belt* described in [6] as the motivating example, which is as follows.

```
#1:report_position {
#2:     p, p.position;       //the package and its position
#3:     r0, r1, r2, r3;   //reader0, reader1, reader2,reader3
#4:     query({r0, r1});
#5:     if (r0.strength ≥r1.strength)
#6:             p.position=0;
#7:     else p.position =1;
#8:     wait(FIXED_INTERVAL);
#9:     query({r0, r1, r2});
#10:    if (r0.strength≥r1.strength and r0.strength≥
          r2.strength)
#11:            p.position=0;
#12:    else if (r1.strength≥r0.strength and r1.strength≥
                r2.strength)
#13:            p.position =1;
#14:    else p.position =2;
#15:    wait(FIXED_INTERVAL);
#16:    query({r1, r2, r3});
#17:    if (r1.strength≥r2.strength and r1.strength≥
          r3.strength)
#18:            p.position=1;
#19:    else if (r2.strength≥r1.strength and r2.strength ≥
                r3.strength)
#20:            p.position=2;
#21:    else p.position=3;
#22:    report p.position;
#23:}

#24:cir_service ϕ = {δ1} {
#25:    δ1 = (q1, s1);
#26:    q1: ∀i>0, ins(p.position)i ≥ ins(p.position)i-1
#27:    s1: drop ins(p.position)i
#28}
```

Figure 1: Pseudo-code and CIR services of Conveyor Belt

Along with a conveyor belt carrying packages to pass through an inspection zone at an airport, four sensors are deployed to partition the inspection zone into a series of segments, and they are labeled as *reader0*, *reader1*, *reader2* and *reader3*, and their positions are denoted as *0*, *1*, *2*, and *3*, respectively. When a package passes through the inspection zone of each sensor, its position is sensed and calibrated according to the position of the sensor that receives the strongest signal strength. Figure 1 shows the pseudo-code that reports the position of the package.

In Figure 1, the package and its position is denoted as *p* and *p.position*, respectively. The readings of four sensors are represented by *r0, r1, r2, r3*, respectively. When the package moves into the inspection zone, the program unit *report_position* will execute and report an estimated package position. The primitive *query*({*c*}) at statement #4, #9, or #16 retrieves the latest context instances from the corresponding context stream. As the sensors may be close to one another, they may induce false positive readings. For instance, a package detected by *r2* via *reader3* may be closest to *reader2*. To tackle such kinds of problem, the program designs one Context Inconsistency Resolution (CIR) service [6] to detect and resolve inconsistent reports of package positions, which is a *context dropping service*. The CIR service specifies the constraint q_1 to assure that *p.position* should be increased monotonically. If q_1 is violated, the CIR service invokes the resolution strategy s_1 to drop the latest *p.position*.

B. Testing Challenges

We use the fault in [6] to motivate our work. The fault is at the statement #10, in which all "≥" are implemented wrongly as "≤". We denote the faulty statement as #10′:

#10′: if (r0.strength ≤ r1.strength and r0.strength ≤ r2.strength)

Using the techniques proposed in [6], testers may construct a test suite that satisfies both the *all-uses* and the *all-services* [6] test data selection criteria. We denote it as *S*:

$$S = \{t_1, t_2, t_3, t_4\}$$

where

$t_1 = \langle(30, 30, 20, 15), (30, 30, 20, 15), (30, 30, 20, 15)\rangle$,
$t_2 = \langle(10, 20, 5, 40), (10, 20, 5, 40), (10, 20, 5, 40)\rangle$,
$t_3 = \langle(30, 20, 35, 10), (30, 20, 35, 10), (30, 20, 35, 10)\rangle$,
and
$t_4 = \langle(20, 30, 35, 40), (25, 33, 38, 40), (25, 33, 35, 40)\rangle$.

To simplify our discussion, let us denote the tuple (*r0*, *r1*, *r2*, *r3*) as a *context variable*, and each of t_1, t_2, t_3, and t_4 is a context stream fragment produced by the four sensors. Each context stream fragment consists of three context instances, each of which corresponds to the sensor readings fed to the program when the *query* primitive is invoked at statements #4, #9, and #16, respectively. For example, the context stream fragment t_4 shows that the context variable changes its values to (20, 30, 35, 40) at #4, then to (25, 33, 38, 40) at #9, and finally to (25, 33, 35, 40) at #16.

TABLE I: TEST OUTCOMES OF THE ALL-USES-ADEQUATE AND THE ALL-SERVICES-ADEQUATE TEST SUITE S*

Test Case	Expected Version		Faulty Version		Context Diversity
	p.position	output	*p.position*	output	
t_1	0→1→1	1	0→1→1	1	0
t_2	1→1→3	3	1→1→3	3	0
t_3	0→2→2	2	0→2→2	2	0
t_4	1→2→3	3	*1→0→3*	3	4

Table I summarizes the test outcome of the test suite *S*. In Table I, the package positions and the output of the program are shown in the "*p.position*" and "*output*" columns, respectively. When t_4 is fed to the faulty version of *Convey Belt*, the positions of the package are reported as 1 at #7, 0 at #10', and 3 at #21, and the final output of the program is 3. (It is noted that, for #10, the context instance violates q_1 and thus is dropped by s_1).

Even though the test suite *S* is both all-uses-adequate and all-services-adequate, yet none of its test cases exposes any failure. To improve the situation, one may enlarge the test suite, or require appropriate test cases satisfying multiple test adequacy criteria. (Indeed, the test suite also satisfies all-edges). Increasing the allowable size of a test suite will incur more test cost because apart from test case selection and execution, the output of the program over the additional test cases need a test oracle to determine whether it is passed or not. It is more attractive if multiple and effective testing criteria can be achieved without affecting the constraint on the size of a test suite. On the other hand, as demonstrated by the example above, *S* satisfying multiple testing criteria may not be effective to reveal the presence of the illustrated faults.

The key technical challenge is to formulate an *effective* criterion that can be used with a wide range of testing criteria. The next section outlines our idea.

IV. OUR SOLUTION

Let us consider another test suite *S*′ that satisfies both the *all-uses* and the *all-services* testing criteria, which is shown as follows:

$$S' = \{t_1', t_2', t_3', t_4'\}$$

where

$t_1' = \langle 30, 20, 5, 10), (30, 40, 5, 10), (30, 40, 5, 40)\rangle$,
$t_2' = \langle(20, 40, 20, 5), (10, 25, 30, 10), (20, 30, 20, 15)\rangle$,
$t_3' = \langle(30, 20, 10, 20), (30, 20, 15, 30), (40, 25, 40, 35)\rangle$,
and
$t_4' = \langle(20, 30, 20, 10), (10, 15, 30, 20), (8, 35, 30, 22)\rangle$.

TABLE II: TESTING OUTCOMES OF S'*

Test Case	Expected Version		Faulty Version		Context Diversity
	p.position	*output*	*p.position*	*output*	
t_1'	0→1→3	3	0→1→3	3	2
t_2'	*1→2→1*	**2**	*1→0→1*	**1**	8
t_3'	0→0→2	2	0→2→2	2	6
t_4'	*1→2→1*	**2**	*1→0→1*	**1**	7

Similar to Table I, Table II summarizes the test results for the test suite *S*′. Table II shows that *S*′ contains two failed test cases (t_2' and t_4'). Although both *S* and *S*′ satisfy the *all-uses* criterion and the *all-services* criterion, they exhibit different testing effectiveness. *S* fails to expose the fault; whereas *S*′ detects the fault twice. What makes such a difference in testing effectiveness of *S* and *S*′?

Suppose that multiple context stream fragments may be feasible to serve as test cases for a testing criterion. Suppose further that all factors related to the internal property of the program under test are equal, we may randomly pick one of such context stream fragments. This is the approach taken by some existing techniques [4][5][6][12].

Let us consider a scenario, in which the self-adaptation of a program occurs when there is a significant change in values within the context streams. If so, different projections (in view of the potential presence of omission faults) of a sequence with relatively high consecutive changes in contextual values may associate with different adaptation sequences. Without further information, the best we can assume is that any projection is feasible and these projections distribute uniformly. We thus measure the overall change in contextual values for a given sequence as *context diversity*:

* We italicize *p.position* sequences which invoke the CIR service ϕ in Table I and Table II.

Definition 4 (Context Diversity): The Context Diversity (CD) of a context stream fragment *cstream*(*C*) is denoted by *CD*(*cstream*(*C*)) and is defined by the following equation:

$$CD(cstream(C)) = \sum_{i=1}^{n-1} HD(ins(C)_i, ins(C)_{i+1}) \quad n = |cstream(C)|$$

where $HD(ins(C)_i, ins(C)_{i+1})$ is the Hamming distance of a pair of context instances $ins(C)_i$ and $ins(C)_{i+1}$, and n is the length of the context stream fragment C.

Take t_4' for example: The Hamming distance between the first pair of two context instances is 4 because the value of *r0* should be edited from 20 to 10 (i.e., one change), and the values for *r1*, *r2*, and *r3* alike. In the same manner, the Hamming distance of the next pair of context instances is only 3 because the value for *r2* does not change. Summing up these two values (3 + 4) gives a value of 7, which is the context diversity value assigned to t_4' as shown in the "Context Diversity" column in Table II. The context diversities for the other test cases are also shown in Table I and Table II.

The measurement metric, context diversity, has at least three desirable properties that we consider interesting:

- Any context stream fragment has a higher (or at least the same) value in context diversity than any of its context stream fragments.
- A more dynamic context fragment receives a higher context diversity value than a less dynamic one if their length are equal.
- Measuring context diversity of a test suite is non-intrusive. Depending on the purpose of testing, a test suite construction method may choose to maximize, randomize, or minimize context diversity in a test suite.

To see the usefulness of context diversity, let us first revisit the *Convey Belt* example. During its execution, the application accepts readings from location sensors as parametric inputs. Moreover, the main component also interacts (via contexts) with a context dropping service to handle (i.e., drop) inconsistent package position values. The removal of value from the context stream may reveal the failure. For instance, from Table II, we observe that execution of the expected version of *Convey Belt* over t_4' produces 2 as the result. It is because the application has triggered the context dropping service, which removes the last position value (which is 1) produced by its main component. This removal exposes the failure caused by #10′. We further observe that the test suite S' has led the application to activate s_1 four times (via t_2' and t_4'); whereas, S results in activating s_1 only once (via t_4). This example illustrates that a test suite with higher value in context diversity may lead the context-aware program to exhibit self-adaptation behaviors more often than a test suite having lower value in context diversity. If producing correct self-adaptation is a major quality assurance problem on top of finding conventional sources of errors, using context diversity may provide a metric value for a testing method to include or exclude a particular test case (that deals with context-adaptation) from a test suite.

Conventional algorithms (CA) for test suite construction are neutral to context diversity. For instance, when constructing an adequate test suite for a test criterion C, given two test cases t and t' which cover the same set of program entities specified by C, many CAs select t and drop t' simply because t has been selected before t' is encountered.

We propose to replace the above-described greedy property in CA by examining the values of context diversity in individual test cases instead. It could be done at the test case level, or at the test suite level. For instance, suppose that the current constructing test suite is X and a new test case x is generated, but x could not improve the coverage of X. Rather than simply dropping x as CA does, our refined algorithm (RA) checks whether it may take out some test cases from X such that the reduced test suite union with $\{x\}$ can achieve the same coverage as X does yet with better (e.g., higher, lower) overall context diversity.

V. DISCUSSION

There are many outstanding questions to be answered: Is it feasible to produce test suites with significantly higher/lower values in context diversity? Does a test suite with better context diversity really improve the fault detection effectiveness of coverage-based testing techniques? What is the time and space cost to measure context diversity? Owing to space constraint, we address the first question in this paper.

TABLE III: DESCRIPTIONS OF TEST SUITES

Test Suite	Context Diversity				Size	Coverage Achieved
	Min	Mean	Max	StdD.		
AS-CA	12.0	**13.3**	14.6	0.570	26	1.000
AS-RA	19.4	**20.6**	21.9	0.481	27	1.000
ASU-CA	12.1	**13.3**	14.4	0.459	43	0.958
ASU-RA	22.4	**22.6**	22.8	0.095	42	0.958
A2SU-CA	11.4	**13.0**	14.3	0.502	50	0.953
A2SU-RA	22.2	**22.5**	22.7	0.091	49	0.950

We have conducted an experiment to validate RA on one application *WalkPath* [6]. In the experiment, we used RA to enhance three testing criteria All-Services (AS), All-Services-Uses (ASU), and All-2-Services-Uses (A2SU) proposed in [6]. Test suites generated by CA with respect to AS, ASU, A2SU are referred to as AS-CA, ASU-CA and A2SU-CA, respectively; they are referred to as AS-RA, ASU-RA, A2SU-RA when RA is used instead. We chose the "the higher, the better" strategy. For each criterion and for each construction algorithm, we independently generate 100 test suites from the same pool of over 20000 test cases.

Table III summarizes some properties of these test suites. The table shows that test suites generated by RA can attain higher overall values in context diversity: ranges of context diversity between AS-CA and AS-RA are non-overlapping. Moreover, the ratio between the means is 1.54. The other two pairs share similar observations. It indicates that using RA, we can construct test suites with significantly changes in the overall context diversity value from using CA.

We are in the process of analyzing the experimental data on the effectiveness and cost-effectiveness of test suites. We

will report them in the future. We are also formulating ways to make context diversity more robust to different types of context-aware and self-adaptive programs.

VI. Related Work

Apart from testing techniques [4][5][6][12] discussed in Section I, there are verification efforts to assure the correctness of context-aware applications. Roman et al. [9] propose Mobile UNITY as a model to represent mobile applications and verify applications against the specified properties. Sama et al. [11] verify the conformance between adaptive behaviors in context-aware applications against a set of proposed patterns. Different from these verification approaches, our work focuses on dynamic testing techniques to assure the quality of context-aware applications. Moreover, the techniques in [4][5][6][12] are white-box in nature; whereas, our measurement metric is a black-box one. Our method complements the above-reviewed work.

The refined algorithm presented in this paper can be viewed as a test case selection problem that aims at changing the level of context diversity of a test suite while satisfying some other criteria. This problem can be modeled as a test case selection problem with multiple goals or constraints [1][7][15]. Nevertheless, the key technical challenge here is how to find out a complementary criterion. Once an effective criterion can be found out, the problem can be further generalized into an optimization with multiple objectives, in which a wide range of algorithms can be used to do the optimization task.

There are also various related papers on considering data values in test case selection or generation. Unlike Korat [2], our technique does not generate test case on its own. Rather, our proposal takes the test case generation capability of existing algorithms for granted. Harder et al. [3] require the generated test suites to cover operational abstractions as many as possible; whereas our technique does not impose additional constraints (with respect with CA) on covering program entities. Netisopakul et al. [8] use white-box structural information to partition loop constructs into several equivalence classes; whereas, our technique uses test case information (rather than the program information).

VII. Conclusion

Context-awareness and context adaptation are two important features of context-aware programs. Although existing testing researches have explored how to use context-related program entities to assure their correctness, they have not explored the potential of contexts within test cases to test self-adaptation behaviors. In this paper, we propose context diversity as a means to measure context stream fragments in terms of their context changes. We also discuss a way to incorporate context diversity in the test suite construction. The preliminary experiment shows that the refined algorithm can improve the context diversities of test suites significantly. We will examine the effectiveness of such test suites and generalize the proposal to handle a wider class of applications, such as event-driven systems.

REFERENCES

[1] J.Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004),* pp. 106–115, 2004.

[2] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002),* pp. 123–133, 2002.

[3] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003),* pp. 60–71, 2003.

[4] Z. F. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesC applications," in *Proceedings of 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE-16),* pp. 94–104, 2008.

[5] 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 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2006/FSE-14),* pp. 242–252, 2006.

[6] H. Lu, W. K. Chan, and T. H. Tse, "Testing pervasive software in the presence of context inconsistency resolution services," in *Proceedings of 30th International Conference on Software Engineering (ICSE 2008),* pp. 100–110, 2008.

[7] N. Mansour and K. EI-Fakin, "Simulated Annealing and Genetic Algorithms for Optimal Regression Testing," *Journal of Software Maintenance: Research and Practice*, 11(1): 19–34, 1999.

[8] P. Netisopakul, L. J. White, J. Morris, and D. Hoffman, "Data coverage testing of programs for container classes," in *Proceedings of 13th International Symposium on Software Reliability Engineering (ISSRE 2002),* pp. 183–194, 2002.

[9] 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 (TOSEM),* 6 (3): 250–282, 1997.

[10] T.H. Tse, S. S. Yau, W.K. Chan, H. Lu, and T.Y. Chen. "Testing context-sensitive middleware-based software applications," in *Proceedings of COMPSAC 2004*, volume 1, pp. 458–465, 2004.

[11] M. Sama, D. S. Rosenblum, Z. M. Wang, and S. Elbaum, "Model-based fault detection in context-aware adaptive applications," in *Proceedings of 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*, pp. 261–271, 2008.

[12] Z. M. Wang, S. Elbaum, and D. S. Rosenblum, "Automated generation of context-aware tests," in *Proceedings of 29th International Conference on Software Engineering* (*ICSE 2007*), pp. 406–415, 2007.

[13] C. Xu, S.C. Cheung, and W.K. Chan, "Incremental consistency checking for pervasive context," in *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pp. 292–301, 2006.

[14] J. Ye, L. Coyle, S. Dobson, and P. Nixon, "Using situation lattices in sensor analysis," in *Proceedings of IEEE International Conference on Pervasive Computing and Communications* (*PerCom 2009*), pp. 1–11, 2009.

[15] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007)*, pp. 140–150, 2007.