To appear in the *International Journal of Software Engineering and Knowledge Engineering*

Integration Testing of Context-Sensitive Middleware-Based Applications: a Metamorphic Approach [*, †, ‡]

W. K. Chan [§]
The Hong Kong University of Science and Technology
wkchan@cse.ust.hk

T. Y. Chen
Swinburne University of Technology
tchen@ict.swin.edu.au

Heng Lu
The University of Hong Kong
hlu@cs.hku.hk

T. H. Tse [¶]
The University of Hong Kong
thtse@cs.hku.hk

Stephen S. Yau
Arizona State University
yau@asu.edu

During the testing of context-sensitive middleware-based software, the middleware checks the current situation to invoke the appropriate functions of the applications. Since the middleware remains active and the situation may continue to evolve, however, the conclusion of some test cases may not easily be identified. Moreover, failures appearing in one situation may be superseded by subsequent correct outcomes and, therefore, be hidden.

We alleviate the above problems by making use of a special kind of situation, which we call checkpoints, such that the middleware will not activate the functions under test. We recommend testers to generate test cases that start at a checkpoint and end at another. Testers may identify relations that associate different execution sequences of a test case. They then check the results of each test case to detect any contravention of such relations. We illustrate our technique with an example that shows how hidden failures can be detected. We also report the experimentation carried out on an RFID-based location-sensing application on top of a context-sensitive middleware.

Keywords: Context-aware application, integration testing, metamorphic testing, RFID testing.

1. Introduction

Context-sensitivity and ad hoc communications [1,16,17,23] are two important properties of ubiquitous computing applications. The former allows applications to detect, analyze, and

† This research is supported in part by a grant of the Research Grants Council of Hong Kong (Project No. HKU 7145/04E) and a grant of The University of Hong Kong.
‡ A preliminary version of this paper was presented at the 5th International Conference on Quality Software (QSIC 2005) (ref. 5).
§ Part of the research was done when Chan was with The University of Hong Kong.
¶ 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 2183. Fax: (+852) 2559 8447. Email: thtse@cs.hku.hk.

react adaptively to changes in attributes, known as *contexts* [13], that characterize the environmental situation. The latter facilitates the components of the applications to communicate dynamically according to the changing contexts.

One kind of ubiquitous computing application is context-sensitive middleware-based software. The middleware is responsible for detecting and handling contexts and situations, with a view to invoking the appropriate local and remote operations whenever any context or situation inscribed in the situation-aware interface is satisfied [23]. Since the applications operate in a situational and highly dynamic environment, this type of configuration increases the intricacy in software quality assurance. Tse et al. [18], for instance, has illustrated through examples the ineffectiveness of common testing techniques, such as control-flow and data-flow testing, which concentrate only on the applications without considering their environment.

Besides, the behaviors of the devices can be so volatile that very complicated mathematics may be required to model the outcomes of an application precisely. As a result, while specifications may exist, it may require a lot of effort to determine the *test oracle* [19], that is, the mechanism against which testers can check a test outcome and decide whether it is correct. The task will become forbidding if there are a large number of test cases.

Under such circumstances, instead of deriving the expected test outcomes from the specification, typical testers would use a weaker means to check the results. They would judge whether a test outcome reveals any failures based on their beliefs or experiences.

There is a growing amount of research aiming at testing ubiquitous computing applications. Axelsen et al. [2] propose a specification-based approach to testing reflective software in an open environment. They model components as algebraic specifications and their interactions as message communication specifications. These specifications will be treated as the test oracle. They suggest using a random selection strategy to produce test cases. When the execution sequence of any test case violates the specifications, it detects a failure. Their approach is essentially an execution monitoring approach.

Flores et al. [9] apply temporal logic to define context expressions in context-sensitive software. They further use an ontological framework to model similar concepts of contexts. These concepts are finally represented as logic predicates. As far as the test case selection strategy is concerned, they apply some form of category partitioning on a custom interface to divide a concept into different partitions. Finally, they propose to have test cases that satisfy the context expressions under the respective predicates. No test case generation method is included. Furthermore, their work does not address the test oracle problem, which is essentially the difficulty in determining the expected outcome of complex software systems such as context-sensitive applications.

Bylund and Espinoza [3] describe how to use a simulation engine to synthesize contexts and exchange them with a context-aware system. From the testing perspective, the simulator generates test cases to a program and receives test outcomes. However, techniques to construct (adequate) test sets [25] or evaluate test outcomes are not discussed. Since then, there have been various researches [11,14,15] proposing different emulators to mimic testing environments of context-aware systems. Still, they do not study how to select effective test cases or how to evaluate them.

Tse et al. [18] propose to construct multiple context tuples as test cases to check whether the outcomes satisfy isotropic properties of context relations. This idea of applying *metamorphic testing* [4,6] is novel. The context tuples are applied to an application function under test atop the context-sensitive middleware. This allows the middleware to detect relevant situations and invoke repeatedly the corresponding functions. The resulting contexts of the test cases are then compared. When there is any discrepancy from an expected context relation, a failure is identified.

However, as we shall discuss in Section 3, if contexts change during test case execution, the technique used in Tse et al. [18] for comparing resulting contexts may overlook a failure. Hence, the technique is applicable only if, during the execution of the test case, (i) the contexts remain static *or* (ii) changes in contexts do not affect any situation expression[a]. To address the situations where neither (i) nor (ii) apply, we study another class of test cases in which sequences of context tuples, instead of isolated context tuples, are used. We shall refer to the new class of test cases as *context-coupled test cases*, and refer to the previous class in Tse et al. [18] as *context-decoupled test cases*. Since transient variations of situations are a major characteristic of context-sensitive middleware-based applications, context-coupled test cases represent an important class of test cases.

This paper proposes a novel approach to integration testing of context-sensitive middleware-based software. For the ease of illustration, we shall use the random testing strategy to generate initial context-coupled test cases.[b] We then use these as source test cases to select follow-up test cases according to the metamorphic relations in question. To facilitate the use of context-coupled test cases for testing, we propose the notion of checkpoints, at which the intermediate and final contexts can be used as test outcomes. Metamorphic relations can then be used to compare the test outcomes of multiple test cases with a view to detecting failures.

In order to evaluate our proposed technique, we conduct experiments on a testbed consisting of a context-sensitive middleware system [21] and its RFID-based location-sensing application that implements the location-estimation algorithm *LANDMARC* [12].

The main contributions of our work are as follows:

(*a*) We extend Tse et al. [18] to address context-coupled test cases, which allow the change of contexts during test case execution.

(*b*) We develop the notion of checkpoints to conduct integration testing, facilitating the checking of test outcomes by the metamorphic testing approach.

(*c*) We recommend practical guidelines for designing test cases for such applications. For example, follow-up test cases should activate context-sensitive function(s) via the middleware at chosen checkpoints. Furthermore, the same test cases should be executed under similar but not necessarily identical environments, so that test outcomes can be compared to reveal failures.

[a] See Section 2.1 for an explanation of situation expressions.

[b] The random generation of initial test cases may be replaced by other test case generation strategies as testers see fit.

(*d*) We evaluate the proposed technique using an RFID-based location-sensing program on a context-sensitive middleware platform.

The rest of the paper is organized as follows: Section 2 introduces the preliminaries necessary for the understanding of our technique. Section 3 discusses the motivations behind our work, develops the notion of checkpoints for testing, and identifies the class of test cases to be examined in this paper. Section 4 illustrates our technique by the example of a smart delivery system. Section 5 describes the experimentation and evaluation of our technique. Finally, a conclusion is given in Section 6.

2. Preliminaries

2.1. *Reconfigurable context-sensitive middleware (RCSM)*

Reconfigurable Context-Sensitive Middleware (RCSM) is a middleware for the ubiquitous computing environment. It supports a Situation-Aware Interface Definition Language (SA-IDL) [22], for specifying context-sensitive application interfaces. Using an SA-IDL specification [22], or SA-spec for short, it provides every application with a custom-made object skeleton that embodies both the context variables and invokable actions. It periodically detects devices in the network, collects raw contextual data from the environment, and updates relevant context variables automatically. Conditions of relevant context values over a period of time are referred to as *situations* [24,22]. Once suitable situations in the SA-spec are detected, the responsible object skeleton will activate appropriate actions.

Using SA-IDL, an SA-spec is defined in terms of three notions. Firstly, SA-IDL adopts an object-oriented context representation to favor the reuse of context data structures. A context is represented as a context class, which is naturally defined in a structure named **RCSM Context Class**. Next, the periodicity to detect and disseminate the context data of a context class is specified in a statement headed by **RCSM Context Acquisition**. Finally, situation expressions are defined in a structure named **RCSM SA Rule**. Details of SA-IDL can be found in Yau et al [22].

A *situation expression* in an SA-spec formulates how to detect situations as well as which action to be activated when a situation is detected. In particular, based on a given SA-spec, the middleware in a device may match a required context variable in its SA-IDL interface with those of surrounding devices. Hence, because of different subgrouping of surrounding devices, the same action of a situation expression may be invoked by the middleware more than once. Each situation expression consists of the keyword, the situation name, the time range, and the situational condition. The keyword **PrimitiveSituation** represents an elementary situation expression, while the keyword **CompositeSituation** means a logical composition of other situation expressions. The time range is of the form $[t, t_0]$ where t_0 is the current time stamp and t is the time stamp at which the situational condition is satisfied.

The mapping relationship between a situation expression and the activated actions is specified through the keyword **ActivateAt**. A "within x" clause in a situation expression asserts that the action will be invoked within x seconds after the situation has been detected.

Similarly, a “frequency = *y*” clause requires the RCSM to probe the contexts at a rate of *y* times per second. A “priority *z*” clause indicates the priority of the action: a higher value of *z* entails a higher priority. We refer to a function used in a situation expression in an SA-spec as an *adaptive function* of the application.

Thus, if a conventional program is seen as a program unit, a context-sensitive middleware-based program extends a conventional program by taking its context-sensitive interfaces into account. Hence, we assume that all interactions of a context-sensitive middleware-based program with its environment are conducted through the context-sensitive application interfaces.

2.2. *Smart delivery system: an example*

In this section, we describe a sample application and illustrate how it can be represented in RCSM. Consider a smart delivery system of a supermarket chain such that individual suppliers replenish their products onto pallets, shelves, and cases in various warehouses according to the demand sent off by such pallets[c]. In the rest of the paper, we shall use the word “pallet” to refer collectively to a shelf, pallet, or case. The smart delivery system includes four features: (i) Each smart pallet can be dynamically configured to store a particular kind of product at, as far as possible, a desired quantity level. (ii) Each van of a supplier delivers a type of goods. (iii) Unsold goods can be returned to the supplier. A smart pallet may request a van to retract certain amount of goods. (iv) The system assumes that the effective delivery distance between any pallet and any van is at most 25 meters.

When a pallet is full, no replenishment is required. When a delivery van moves along a street, a particular pallet may detect the van and request for replenishment if the desired quantity is not met. If there are enough goods in the van, the request is entertained.

The replenishment signal may also be sensed by any other delivery van(s) nearby. A van may not be able to deliver the requested quantity of goods to a particular pallet, however, if there are other pallets requiring replenishment. Because of the interference among vans, possibly from different suppliers, and the presence of other nearby pallets with the same goods, the actual amount of goods in a pallet may differ from its desired level.

Figure 1 shows a sample situation-aware interface specification for the device in delivery vans. We have simplified the SA-IDL specification by assuming that a van will deliver the same amount of goods to requesting pallets in each round of delivery. We have also assumed only one type of product.

The situation *understock* represents that, when the pallet is inside the effectively delivery region at time *t* of the received context, the current ledger amount q_l at the pallet site[d] has been short of the desired quantity q_d for more than a tolerance of ε for the last 3 sec-

[c] Readers may be interested to read the press release that “Wal-Mart has set a January 2005 target for its top 100 suppliers to be placing RFID [radio frequency identification] tags on cases and pallets destined for Wal-Mart stores ...” It is emphasized that “the first to market wins”.

[d] A ledger amount includes the quantity of goods in a particular pallet as well as the quantity of goods that a van wishes to add to the pallet. In this paper, whenever there is no ambiguity, we simply use q_l instead of *Pallet*.q_l. This kind of simplification applies to all context variables.

```
#define ε 5
RCSM Context Class Van extends Base {
    float $q_v$;          // the quantity of goods deliverable by the van
    Position $p_v$;       // the location of the van in $(x, y)$ coordinates
    float $d$;            // square of distance between the van and a pallet
}
RCSM Context Class Pallet extends Base {
    int $s$;              // no. of vans surrounding the pallet
    float $q_d$;          // the desired quantity of goods for the pallet
    float $q_l$;          // the ledger amount of goods in the pallet
    float $q_p$;          // the quantity of goods on hand in the pallet
    Position $p_p$;       // the location of the pallet in $(x, y)$ coordinates
}
RCSM Context Acquisition {Pallet {frequency = 1;}}
RCSM SA Rule $Smart_{Van}$ {
    Derived Van.d
      $(Van.p_v.x - Pallet.p_p.x)^2 + (Van.p_v.y - Pallet.p_p.y)^2$
    PrimitiveSituation overstock
      $([-3, 0]\,(Pallet.q_l - Pallet.q_d > \varepsilon) \wedge (d \leqslant 625))$;
    PrimitiveSituation understock
      $([-3, 0]\,(Pallet.q_d - Pallet.q_l > \varepsilon) \wedge (d \leqslant 265))$;
      // Note: 625 is written as 265 by mistake

    ActivateAt overstock {
      [local] void withdraw() [within1] [priority1]}
    ActivateAt understock {
      [local] void replenish() [within1] [priority1]}
}
```

Fig. 1. A simplified SA-IDL specification for the smart device in delivery vans.

onds. When this is the case, the application would like to replenish the goods in the pallet. This is accomplished by invoking the local function *replenish*( ). A situation *overstock* is similarly defined.

There is an error in the SA-IDL specification of the device in delivery vans in Figure 1. In the situation expression *understock*, the value "625" is written as "265" by mistake.

The functions *replenish*( ) and *withdraw*( ) are used to supply or retract goods. They increment and decrement the context variable q_v by 1, respectively, and both operations are executed non-deterministically. The middleware invokes the functions a number of times to achieve the required delivery amount. The overall ledger amount at the pallet site may oscillate, sometimes higher than the desired quantity and sometimes lower, and will eventually reach the desired value.

Figure 2 shows a correct implementation of the functions *replenish*( ) and *withdraw*( ). Once a new value for the context variable q_v is computed, it should be detected by the middleware at the pallet site. This example assumes that there is a correct test stub for the function *computeLedgerAmount*( ) in the pallet device to take the values of q_v from all the surrounding vans and to compute a corresponding new value for the context variable q_l. The theoretical formula to compute the variable q_l is defined as follows, although tolerances such as $|q_l - q_d| < \varepsilon$ may need to be added in the real-life implementation:

$$q_l = \sum_{i=1}^{s} q_v^{(i)} + q_p$$

where $q_v^{(i)}$ denotes the context variable q_v from the i-th surrounding van. For a configuration with only one pallet and one van, the formula can be simplified to:

$$q_l = q_v + q_p \tag{2}$$

2.3. *Metamorphic testing*

Metamorphic testing[4,6,8] is a property-based testing approach. It is based on the intuition that, even if a test case does not reveal any failure, follow-up test cases should be constructed to check whether the software satisfies some necessary conditions of the target solution of the problem. These necessary conditions are known as metamorphic relations.

Let f be a target function and let P be its implementation. Intuitively, a metamorphic relation is a necessary condition over a series of inputs $x_1, x_2, \ldots, x_n$ and their corresponding results $f(x_1), f(x_2), \ldots, f(x_n)$ for multiple evaluations of f. This relation must be satisfied when we replace f by P; otherwise P will not be a correct implementation of f.

Metamorphic relation and metamorphic testing can be formally defined as follows:

Definition 1. (Metamorphic Relation) Let $\langle x_1, x_2, \ldots, x_k \rangle$ be a series of inputs to a function f, where $k \geq 1$, and $\langle f(x_1), f(x_2), \ldots, f(x_k) \rangle$ be the corresponding series of results. Suppose $\langle f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}) \rangle$ is a subseries, possibly an empty subseries, of $\langle f(x_1), f(x_2), \ldots, f(x_k) \rangle$. Let $\langle x_{k+1}, x_{k+2}, \ldots, x_n \rangle$ be another series of inputs to f, where $n \geq k+1$, and $\langle f(x_{k+1}), f(x_{k+2}), \ldots, f(x_n) \rangle$ be the corresponding series of results. Suppose, further, that there exists relations $r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n)$ and $r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$ such that r' must be true whenever r is satisfied. We say that

$$\begin{aligned} \boldsymbol{MR} = \{\ & (x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \mid \\ & r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n) \\ & \rightarrow r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))\ \} \end{aligned}$$

is a metamorphic relation. When there is no ambiguity, we simply write the metamorphic relation as

$$\begin{aligned} \boldsymbol{MR}\text{: } & \text{If } r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n) \\ & \text{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)). \end{aligned}$$

Furthermore, x_1, x_2, …, x_k are known as the *source test cases* and x_{k+1}, x_{k+2}, …, x_n are known as the *follow-up test cases*.

Definition 2. (Metamorphic Testing) Let P be an implementation of a target function f. The *metamorphic testing* of metamorphic relation

$$\begin{aligned} \boldsymbol{MR}\text{: } & \text{If } r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n), \\ & \text{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \end{aligned}$$

involves the following steps: (1) Given a series of source test cases $\langle x_1, x_2, \ldots, x_k \rangle$ and their respective results $\langle P(x_1), P(x_2), \ldots, P(x_k) \rangle$, generate a series of follow-up test cases $\langle x_{k+1}, x_{k+2}, \ldots, x_n \rangle$ according to the relation $r(x_1, x_2, \ldots, x_k, P(x_{i_1}), P(x_{i_2}), \ldots, P(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n)$ over the implementation P. (2) Check the relation $r'(x_1, x_2, \ldots, x_n, P(x_1), P(x_2), \ldots, P(x_n))$ over P. If r' is false, then the metamorphic testing of *MR* reveals a failure.

Consider a program which, for any given x-coordinate as input, computes the y-coordinate of a straight line that passes through a given point (a, b). A sample metamorphic relation is

$$\textbf{\textit{MR'}}\text{: If } x_2 - a = k(x_1 - a), \text{ then } f(x_2) - b = k(f(x_1) - b).$$

Suppose the given point is $(3, 4)$, and suppose the source test case $x_1 = 5$ produces $P(x_1) = 7$. We can identify a follow-up input, say $x_2 = 8$. If the program produces $P(x_2) = 11$, then the metamorphic relation is violated. This signals a failure.

Throughout the course of checking results in metamorphic testing, there is no need to predetermine the expected result for any given test case, such as whether $P(5)$ should be the same as the test oracle $f(5)$, and whether $P(8)$ should be 11.5. Since there is no need to check the results of test cases through other means such as a formal test oracle, it alleviates the test oracle problem [e]. Further implementations and applications of metamorphic testing have been reported, for example, by Gotlieb and Botella [10] and Wu [20].

3. Checkpoints in Context-Sensitive Middleware-Based Applications

3.1. *Motivations*

Let us firstly consider the motivations for enhancing the testing technique proposed in our previous work [18]. Given a scenario with one van and one pallet, the tester may generate the following two test cases:

$$u_1 = (s = 1, q_d = 100, q_l = 50, q_p = 0, q_v = 7, p_p = (1, 1), p_v = (0, 0))$$
$$u_2 = (s = 1, q_d = 100, q_l = 73, q_p = 73, q_v = 7, p_p = (10, 20), p_v = (4, 4))$$

Consider the test case u_1. Initially, the middleware detects that the condition *understock* is satisfied and, hence, invokes the function *replenish*( ) to increment q_v by 1. The detection of *understock* will continue until q_l gradually reaches 95. At this point, the difference between q_d and q_l is $100 - 95$, which is no more than the tolerance limit $\varepsilon = 5$. As for the test case u_2, we have $d = (10-4)^2 + (20-4)^2 = 292$. Since $d > 265$, the middleware does not detect an *understock* situation. The test stub *computeLedgerAmount*( ) will update q_l to 80 according to Equation (2). Thus, the following context tuples will result:

$$CT_{u1} = (s = 1, q_d = 100, q_l = 95, q_p = 0, q_v = 95, p_p = (1, 1), p_v = (0, 0))$$
$$CT_{u2} = (s = 1, q_d = 100, q_l = 80, q_p = 73, q_v = 7, p_p = (10, 20), p_v = (4, 4))$$

[e] This paper serves as an illustration of how metamorphic testing can be usefully applied in the integration testing of context-sensitive middleware-based applications. We shall not address the principles and procedures of formulating metamorphic relations. We refer readers to Chen et al. [8] for relevant discussions on metamorphic testing.

```
void replenish(){                 void withdraw(){
s1  int r;                        s7   int r;
s2  r = rand() % s;               s8   r = rand() % s;
    // randomize the action            // randomize the action
s3  if r == 0 {                   s9   if r == 0 {
s4     if q_v < MAX {             s10     if q_v > 0 {
s5        q_v = q_v + 1;          s11        q_v = q_v − 1;
    }}                                 }}
s6  sleep(r/2);                   s12  sleep(r/2);
}                                 }
```

Fig. 2. Implementation of *replenish*( ) and *withdraw*( )

Our previous work suggests testing against metamorphic relations such as “when the distances between the pallet and the van for both test cases are comparable, the ledger quantities q_l for both test cases should also be comparable.” Since the corresponding values of q_l (95 versus 80) in CT_{u_1} and CT_{u_2} do not agree, the metamorphic relation is violated and, hence, a failure is revealed. While the proposal to bypass complicated test oracles by checking isotropic properties is innovative, there are a few limitations:

Firstly, our previous work does not deal with changes in contexts during a test case execution. It is assumed that the contexts are fixed or can be ignored once a test execution starts. The previous assumption is not without good practical reasons. When both a pallet device and the device in a delivery van are mobile in arbitrary speeds and directions, it is difficult for a tester to find a complex mathematical model to represent their motions and to generate follow-up test cases. In the present paper, we propose to relax the assumption and address this difficult problem.

Secondly, in the smart delivery system, a van may move and a pallet may be relocated, so that the distance between a van and a pallet may change. Since the middleware always remain active, the original situation that triggers a test case may not apply throughout the period of its execution. Failure that occurs at a certain instant may be hidden again at the conclusion of a test case execution. This can be illustrated as follows:

When the distance between a van and a pallet *again* falls within the activation distance, such as 16.27 meters, the adaptive function *replenish*( ) will be activated a second time by the middleware. When the devices are kept within the activation distance for a sufficiently long period of time, multiple activations of *replenish*( ) will result. This will change the ledger amount q_l at the pallet site to the desired quantity q_d within the tolerance limit ε as stated in the situation-aware interface. Consider, for example, u_2 again. Suppose that testers reduce the separation distance to 16.27 meters after the context tuple CT_{u_2} above has been computed. After 15 successful increments of q_v by the function *replenish*( ), the context tuple will become

$$CT'_{u2} = (s = 1, q_d = 100, q_l = 95, q_p = 73, q_v = 22, p_p = (4, 0), p_v = (4, 16.27))$$

As a result, the failure that should be revealed by CT_{u_2} is actually hidden when the test case terminates. Detecting failures based on the final contexts of a test case is, therefore, more difficult in context-sensitive middleware-based applications than the conventional counterparts. This will also need to be addressed.

Thirdly, as the middleware remains active and situation may continue to evolve, the termination of some test cases may not be easily identified. We propose to use a new concept of "checkpoints" in lieu of the detection of termination.

3.2. *Checkpoints*

We recall that an environmental situation is characterized by a set of contexts that may change over time. In order to detect a relevant situation via situation expressions, the middleware will need to activate adaptive functions, as explained in the last paragraph of Section 2.1.

There are circumstances where none of the situation expressions are relevant to the adaptive functions under test. For such situations, the middleware will not activate any adaptive function. We refer to these situations as *checkpoints*.

Let us give an illustration of a checkpoint using the example in Section 3.1. Suppose the input u_1 is applied to the function *replenish*( ). After a few rounds of activations of *replenish*( ), the application produces the context tuple CT_{u_1}. We can observe from Figure 1 that no further function activation will be possible unless the context is changed by some external factor. Hence, a checkpoint has been reached.

To identify checkpoints, we may, for example, negate every situational condition (excluding the temporal part) to form a constraint equation, and find the roots of the set of equations as the checkpoints. Consider Figure 1 again. There are two situational conditions, namely

$$\begin{aligned} s_1\colon\ & [-3,0]\,(\mathit{Pallet}.q_l - \mathit{Pallet}.q_d > \varepsilon) \wedge (d \leq 625) \\ s_2\colon\ & [-3,0]\,(\mathit{Pallet}.q_d - \mathit{Pallet}.q_l > \varepsilon) \wedge (d \leq 265) \end{aligned}$$

Their respective negation forms are:

$$\begin{aligned} n_1\colon\ & [-3,0]\,(\mathit{Pallet}.q_l - \mathit{Pallet}.q_d \leq \varepsilon) \vee (d > 625) \\ n_2\colon\ & [-3,0]\,(\mathit{Pallet}.q_d - \mathit{Pallet}.q_l \leq \varepsilon) \vee (d > 265) \end{aligned}$$

Any context tuple satisfying both of the conditions n_1 and n_2 will form a checkpoint. [f]

By treating checkpoints as the starting and ending points of a test case, they provide a natural environmental platform for the integration testing of the functions of a system. This setting offers an opportunity to test the functions in different parts of the application within the same environment. When a test case is being executed, the situation of the functions under test may change. The changing situation may or may not represent checkpoints of other functions *not* under test, depending on whether situation expressions of the latter functions are inert to these changes. Detailed discussions on the design of a non-interference test setup are beyond the scope of this paper. Nonetheless, we note that when the changing situation of a test case happens to represent checkpoints of functions *not* under test, there is no need to apply auxiliary testware to neutralize the ripple effects of the contexts on the rest of the application.

[f] We are aware of the difficulty of the constraint satisfaction problem, which deserves further exploration in future research.

We shall explore in detail the testing techniques related to the application of checkpoints to context-coupled test cases in Sections 3.3 and 4. We note a couple of practical considerations before applying the concept. Firstly, a middleware may depend on the current contexts as well as the historical contexts to determine an activation situation. Testers may have to determine from the limited execution history of a test case whether the middleware will finally activate some adaptive function. In theory, this may not be feasible. In practice, however, as there are "within x" clauses defined in SA-IDL specifications, an RCSM-like middleware provides a bounded waiting period for testers to conclude whether a checkpoint has been reached. Secondly, in general, an application may or may not have checkpoints. In this paper, we shall limit our discussions to applications that will reach some checkpoints. On the other hand, the formulation of guidelines for the selection of different checkpoints is both a testing criterion problem and a refined oracle problem and deserves further investigation in future work.

3.3. *Test cases at checkpoints*

When a middleware reaches a checkpoint, a further change in context may or may not trigger the middleware to activate adaptive functions under test. There are three possible cases.

Case (1): Test case has reached a final checkpoint. In other words, there is no possibility of further activation of functions. The collection of context tuples (or contexts for short) represents a *final* checkpoint of the application. Verifying whether it is a valid combination of contexts for the application can be performed.

In general, the contexts of a final checkpoint may or may not be observable. Suppose, for sake of argument, that they are observable. We may compare the results with those of another test case that has also reached a checkpoint according to some metamorphic relation such as the isotropic property on the context q_l illustrated in Section 3.1. Of course, if it is not possible to observe the context results of the application, it will be an open verification problem, which will not be addressed in this paper.

Case (2): Test case will reach another checkpoint. In other words, the middleware will activate some functions and then reach another checkpoint.

Obviously, none of the situation expressions are satisfied at the checkpoint; otherwise the middleware would continue to activate further functions. Hence, checking the contexts against the situation expressions as post-conditions is ineffective. Having said that, research shows that, given a relation, a derived relation may have a better fault detection capability than the given one [7].

Although we stated earlier that we would not address the principles of formulating metamorphic relations in this paper, we would like to add that some useful expected relations can be derived from situation expressions. Testers may then confirm whether such relations are indeed expected relations. Take the situational condition *overstock* as an example. Suppose there are two test cases, t_1 and t_2, that result in some checkpoints. Suppose the contexts d, q_d, and q_l of the test case t_i are denoted by d_i, q_{d_i}, and q_{l_i}, respectively. For either checkpoint, *overstock* does not hold; otherwise the middleware would activate the function *withdraw*( ) further. Hence, we have $(q_{d_1} - \varepsilon \leqslant q_{l_1} \leqslant q_{d_1} + \varepsilon) \wedge d_1 \leqslant 625$ and $(q_{d_2} - \varepsilon \leqslant q_{l_2} \leqslant q_{d_2} + \varepsilon) \wedge d_2 \leqslant 625$. [g]

In general, there are 3 possible relations between q_{d_1} and q_{d_2}, namely, "$=$", "$<$", and "$>$". When $q_{d_1} = q_{d_2}$, if the pallet(s) for both test cases are within the delivery distance, we must have $|q_{l_1} - q_{l_2}| \leqslant 2\varepsilon$, or $q_{l_1} \approx q_{l_2}$ for short. Substituting it into the above equation, we have:

***MR*₁**: If $q_{d_1} = q_{d_2}$, $d_1 \leqslant 625$, and $d_2 \leqslant 625$, then $q_{l_1} \approx q_{l_2}$.

Similarly, we can derive appropriate relations for the cases where $q_{d_1} < q_{d_2}$ and $q_{d_1} > q_{d_2}$.

Obviously, if test cases are context-decoupled, subsequent values of d_1, d_2, q_{d_1}, and q_{d_2} are expected to be unchanged or can be ignored during the executions of the two test cases. This will result in a metamorphic relation similar to $MR_{PowerUp}$ in Tse et al. [18]

For context-coupled test cases, checking the metamorphic context relations may not reveal failures, as discussed in Section 3.1. In this paper, we propose to test relations of multiple test execution sequences, similarly in style to metamorphic context relations but more complex in detail.

Let us consider the test case u_1 in Section 3.1. During the execution of u_1, a number of *replenish*( ) function invocations are expected to occur. Each of them will increment the context variable q_v by 1. Similarly, the function *withdraw*( ) is expected to decrement q_v by 1. The test case u_1 originally invokes the function *replenish*( ) 88 times to reach the context CT_{u_1}. Suppose u'_1 is a follow-up test case that has the same behaviors, but an additional *withdraw*( ) is called before u'_1 is executed. Then, u'_1 will invoke *replenish*( ) 89 times instead of 88.

Since the functions *replenish*( ) and *withdraw*( ) are symmetric in nature, we can generalize the situations and formulate the following metamorphic relation:

> ***MR*₂**: Let t be a source test case and t' be a follow-up test case that share the same checkpoint, known as an *initial checkpoint*. If we apply *withdraw*( ) to the initial checkpoint before executing t', then the number of invocations of the *replenish*( ) function for t' is expected to be more than that of t. If we apply *replenish*( ) to the initial checkpoint before executing t', then the number of invocations of the *withdraw*( ) function for t' is expected to more be than that of t.

Case (3): Test case will not reach another checkpoint. In other words, the middleware will activate functions repeatedly and will not terminate. Since the system does not terminate, it may already represent a failure. On the other hand, if non-terminating invocations do not mean a failure, testers may propose metamorphic relations between the context sequences of two test cases, similarly to Case (2) above.

We note that, in general, termination is undecidable. Testers may not be able to distinguish Case (3) from Case (2). In practice, testers may regard the software to have terminated after some maximum period of time has elapsed. They can collect the statistics, such as the mean values, of the contexts over a period of time as the resulting contexts. In this way, Case (3) will degenerate to Case (2). For the ease of discussions, we shall restrict ourselves to only Case (2) in this paper.

[g] Without loss of generality, we assume that all variables carry positive values in the illustration.

3.4. *The same test case in distinct but similar environments*

Apart from using different test cases under isotropic environments to test a program, testers may also consider using the same test case under distinct but similar environments for metamorphic testing. In location estimation, for example, the testing environment of a context-aware location-sensing program is difficult to emulate because the actual environment, captured as contexts, may vary slightly from time to time. Testers can compare the relative changes between corresponding checkpoints of the same test case to detect any anomaly in test outcomes. We shall give an illustrative example in Section 4. This will also be analyzed in detail in Section 5.

4. Example of Context-Coupled Test Case with Follow-Up Test Case

In the last section, we have identified a new class of context-coupled test cases that remain unexplored in our previous work, and introduced a new concept of checkpoints with a view to revealing failures using such test cases. In this section, we apply the concepts to detect the failures caused by the fault in Figure 1, that is, the condition $d \leqslant 265$ instead of $d \leqslant 625$ in the situation *understock.*

We shall use a notation different from previous sections to accommodate the features of a context-coupled test case. We define a test case t in two parts, namely, the initial context tuple CT_t and a sequence $\vec{\Theta}_t$ of context updates. The first element in $\vec{\Theta}_t$ comes from CT_t.

Definition 3. (Context-Coupled Test Case) A *context-coupled test case t* is a tuple $(CT_t, \vec{\Theta}_t)$, such that CT_t is a n-tuple representing the initial context and $\vec{\Theta}_t$ is a series of context updates $\langle \theta_t[0], \theta_t[1], \ldots, \theta_t[k] \rangle$, where each $\theta_t[0]$, $\theta_t[1]$, ..., $\theta_t[k]$ is an m-tuple ($m \leq n$) representing the values of a set of context variables at time $t[i]$, $t = 0, 1, \ldots, k$. Elements of every $\theta_t[i]$ are of the format $\langle$*context variable*$\rangle = \langle$*value*$\rangle$ such that no two context variables in the elements of $\theta_t[i]$ are identical. Each element in $\theta_t[0]$ comes from CT_t.

For the ease of illustration, "nice-looking" numerical values without decimal places are used in the examples.

4.1. *Context-coupled test case t_1*

Consider a context-coupled test case t_1 below for testing the configuration of one pallet device and one device in a delivery van, with a test stub *computeLedgerAmount*( ) in the pallet device. Following the nomenclature in metamorphic testing, we shall refer to it as the source test case.

$$\begin{aligned}
t_1 &= (\ CT_{t_1}, \vec{\Theta}_{t_1}\ ) \\
CT_{t_1} &= (\ s = 1, q_d = 20, q_l = 20, q_p = 8, q_v = 12, p_p = (17, 1), p_v = (1, 1)\ ) \\
\vec{\Theta}_{t_1} &= \langle\ (\boldsymbol{d = 256},\ \boldsymbol{q_d = 20},\ \boldsymbol{q_p = 8}),\ (d = 240, q_d = 30, q_p = 12), \\
&\quad (d = 210, q_d = 33, q_p = 12),\ (d = 300, q_d = 18, q_p = 18), \\
&\quad (d = 320, q_d = 22, q_p = 15),\ (\boldsymbol{d = 200},\ \boldsymbol{q_d = 22},\ \boldsymbol{q_p = 15}), \\
&\quad (d = 180, q_d = 20, q_p = 18),\ (d = 170, q_d = 19, q_p = 16), \\
&\quad (d = 120, q_d = 18, q_p = 15),\ (d = 80, q_d = 18, q_p = 17), \\
&\quad (d = 20, q_d = 19, q_p = 22),\ (d = 30, q_d = 22, q_p = 21), \\
&\quad (\boldsymbol{d = 30},\ \boldsymbol{q_d = 22},\ \boldsymbol{q_p = 21})\ \rangle
\end{aligned}$$

Table 1. Updated contexts for test cases t_1 and t_2.

No.	d	q_d	q_v	q_p	q_l
1	256	20	12	8	20
2	240	30	12	12	24
3	210	33	13	12	25
4	300	18	14	18	32
5	320	22	13	15	28
6	***200***	***22***	***12***	***15***	***27***
7	180	20	12	18	30
8	170	19	11	16	27
9	120	18	10	15	25
10	80	18	9	17	26
11	20	19	8	22	30
12	30	22	7	21	28
13	30	22	6	21	27

(a) Updated contexts for t_1

No.	d	q_d	q_v	q_p	q_l
1	256	20	12	8	20
2	240	30	12	12	24
3	210	33	13	12	25
4	300	18	14	18	32
5	320	22	13	15	28
6	200	22	12	15	27
7	200	22	***12***	15	27
8	180	20	12	18	30
9	170	19	11	16	27
10	120	18	10	15	25
11	80	18	9	17	26
12	20	19	8	22	30
13	30	22	7	21	28
14	30	22	6	21	27

(b) Updated contexts for t_2

Source test cases may be generated randomly, or through conventional black- or white-box approaches.

Step (1): Apply the initial context CT_t to the one pallet and one van configuration. Update the derived context d to 256.[h] According to Equation (2), the test stub *computeRadiance*( ) will change q_l to $12+8=20$. Since q_l and q_d are 20, according to situation expressions *overstock* and *understock* in Figure 1, the middleware will not be triggered to activate any function. The application is, therefore, at a checkpoint.

Step (2): Apply the second context update of $\vec{\Theta}_{t_1}$ (that is, $(d=240, q_d=30, q_p=12)$) to the configuration. One possible way to enable the required context update is to set the desired quantity q_d of the pallet device to 30, move the pallet device from location coordinate $(17,1)$ to $(\sqrt{240},1)$, and add 8 unit of goods to this particular pallet. The test stub will update q_l from 20 to $12+12=24$. The updated contexts of the configuration are shown in Table 1.

Step (3): Since the difference between q_d and q_l is greater than the tolerance limit $\varepsilon = 5$, and since d is not more than 265, the situation *understock* is detected and, hence, *replenish*( ) is invoked by the middleware. The context variable q_v is updated from 12 to 13 by *replenish*( ). This is an automatic step.

Step (4): Testers then apply the third context update of $\vec{\Theta}_{t_1}$ (that is, $(d=210, q_d=33, q_p=12)$) to the configuration. This changes q_l from 24 to 25.[i]

Step (5): The above interleaving of context updates by testers and automatic activations of functions by the middleware continues for 3 more rounds. Testers have applied the 6th entry of $\vec{\Theta}_{t_1}$. The context variable q_l will be 27 after the function *withdraw*( ) has decre-

[h] Since the situation expressions in Figure 1 deal directly with the derived context d, we shall refer to d instead of the basic contexts p_p and p_v for the ease of discussion.

[i] A convenient way to enable such timely updates is to emulate the device using a device simulator, so that the temporal constraints can be controlled by testers.

mented q_v from 13 to 12. Comparing the context variables q_l and q_d, no situation inscribed in the situation interface is fulfilled. The configuration has reached a checkpoint. Instead of waiting for a further activation by the middleware, therefore, testers apply the 7th context update ($d = 180$, $q_d = 20$, $q_p = 18$). Steps (2)–(5) are then repeated for the rest of $\vec{\Theta}_{t_1}$.

Step (6): Finally, the test case execution reaches the 13th entry of $\vec{\Theta}_{t_1}$.[j] It completes the execution of the interactive test case t_1. The test case t_1 will decrement q_v gradually after the 4th entry in $\vec{\Theta}_{t_1}$. This is done either by invoking the function *withdraw*( ) or, in case that a checkpoint has been reached, by retaining the previous value for q_v.

Since we are interested in applying other adaptive functions to a selected checkpoint of the source test case as discussed in Section 3.3, the three context updates that will result in checkpoints of the application configuration are highlighted in $\vec{\Theta}_{t_1}$. For the same reason, testers may randomly generate a source test case t_1 as long as they can find checkpoints during its execution.

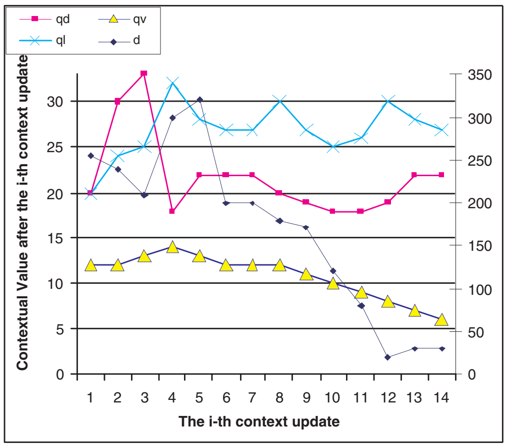
4.2. *Follow-up test case t_2*

Following the concepts presented in Case (2) of Section 3.3, a follow-up test case t_2 of t_1 should share the same initial checkpoint as t_1. Firstly, testers should identify a checkpoint in t_1. As highlighted in $\vec{\Theta}_{t_1}$, there are several possible choices. Suppose testers choose the second checkpoint, namely, the 6th entry in $\vec{\Theta}_{t_1}$. For the ease of description, we shall denote it by S. According to MR_2, testers would like to provide a situation S' consistent with S such that (i) it expects to invoke the adaptive function *replenish*( ), and (ii) it increases the number of subsequent invocations of the adaptive function *withdraw*( ).

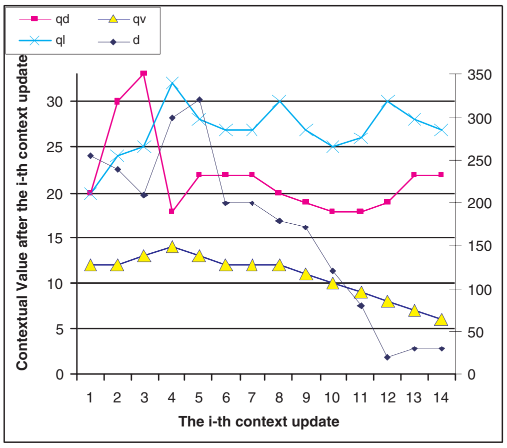
There are many methods to set up S'. One approach is to use an auxiliary pallet device. For instance, testers may use the pallet device of test case u_2 in Section 3.1, namely, $p_2 = (s' = 1, q'_d = 100, q'_n = 73, q'_o = 73, p'_v = (10, 20))$. According to the description in Section 2.2 and the SA-IDL specification in Figure 1, the replenishment request will be triggered in 4 seconds. This auxiliary pallet device is, therefore, expected to join the network at situation S for 4 seconds, and then leave the network. Afterward, the rest of the test case t_1 (that is, the context variables d, q_d, and q_p of the 7th to 13th entries of $\vec{\Theta}_{t_1}$) is applied as scheduled. The test case t_2 is as follows:

$$\begin{aligned}
t_2 &= (\ CT_{t_2}, \vec{\Theta}_{t_2}\ ) \\
CT_{t_2} &= (\ s = 1, q_d = 20, q_l = 20, q_p = 8, q_v = 12, p_p = (17, 1), p_v = (1, 1)\ ) \\
\vec{\Theta}_{t_2} &= \langle\ (d = 256, q_d = 20, q_p = 8),\ (d = 240, q_d = 30, q_p = 12), \\
&\quad (d = 210, q_d = 33, q_p = 12),\ (d = 300, q_d = 18, q_p = 18), \\
&\quad (d = 320, q_d = 22, q_p = 15),\ (d = 200, q_d = 22, q_p = 15), \\
&\quad (\boldsymbol{d = 200,\ q_d = 22,\ q_p = 15}),\ (d = 180, q_d = 20, q_p = 18), \\
&\quad (d = 170, q_d = 19, q_p = 16),\ (d = 120, q_d = 18, q_p = 15), \\
&\quad (d = 80, q_d = 18, q_p = 17),\ (d = 20, q_d = 19, q_p = 22), \\
&\quad (d = 30, q_d = 22, q_p = 21),\ (d = 30, q_d = 22, q_p = 21)\ \rangle
\end{aligned}$$

[j] After all context updates have been applied, the middleware may still detect situations and, hence, may invoke functions until the configuration reaches a checkpoint. Without the loss of generality, we assume that the test case will reach a checkpoint immediately after the final context update in $\vec{\Theta}_{t_1}$.



(a) Actual



(b) Expected

Fig. 3. Context trends for test case t_2.

We firstly verify the results at checkpoints. Since the metamorphic relation MR_1 is applicable, testers may apply it for testing. However, as discussed in Section 3.1, the failure is subtle. It occurs immediately after the application of the situation S' to the test configuration. The context variable q_v should be decremented, but is actually not. Owing to the subsequent detections of the *overstock* situation followed by *replenish*( ) actions, the next checkpoint of the test case leaves no footprint of the failure. In short, MR_1 cannot reveal any failure.

On the other hand, both test cases have same number of *withdraw*( ) invocations (related to entries 7–13 for test case t_1 and entries 8–14 for t_2) between the second checkpoint and the final one. This violates relation MR_2, and hence, reveals a failure.

Interested readers may wish to know whether it is easy to recognize the failures via other means, such as by comparing the resulting context values with the expected values. Context d is plotted against the y-axes on the right of these graphs. All the other contexts are plotted against the y-axes on the left. Figure 3(b) shows the expected results of test case t_2 in a correct implementation. The two charts look remarkably similar. Since the fault only causes the value of q_v to be updated once, the failure is quite subtle. In short, our technique helps testers identify failures that may easily be overlooked.

4.3. *The same test case in distinct but similar environments*

Suppose the smart delivery system is extended with a location-estimation feature to allow delivery vans to check their positions. Ideally, a van should detect and report going through the same path if they travel along the same route at different times of the day. In real life, however, there may be small variations in the environment at different time slots. In the other hand, the variations in the detected paths are expected to be small even in such situations. Testers can then use this expectation as a metamorphic relation to test the system with the same test case going through two distinct but similar environmental conditions. In the next section, we will describe our experimentation on the identification of failures using this technique.

5. Experimentation

The effectiveness of failure detection using different test cases under isotropic environments is easy to illustrate, as we have done in Section 4. On the other hand, the effectiveness using the same test case under distinct but similar environments is more difficult to demonstrate. In this section, we report on the experimentation of an RFID-based location-estimation program running on a context-aware middleware prototype [21]. The sample application is based on the example discussed in Section 4.3. We apply our technique of comparing the test outcomes of the same test case in similar environments. We execute the same set of test cases on the prototype application and its faulty versions during two different time slots in the pervasive laboratory of the Hong Kong University of Science and Technology. We shall refer to the environmental settings of the two time slots as Environments E_1 and E_2, respectively.

In summary, our approach identifies 71.4% of faulty versions. A control experiment using an intuitive benchmark on the same set of test cases in Environment E_1 reveals only 33.3% of the faulty versions, while another experiment in Environment E_2 reveals only 38.1%. Thus, the experimentation indicates that our approach is promising, even though we need to increase the testing effort by running the same test case under distinct environments. The details are as follows.

5.1. *Setup*

The testbed. Our experimentation is carried out on *Cabot* system 2.0 [21] and its evaluation application, which implements the *LANDMARC* RFID-based location-sensing algorithm [12]. The contexts and situations are represented in XML. All other parts are implemented in Java. Our experiment focuses on the testing of the application, and assumes that the middleware is correct.

The basic feature of the application is as follows: The middleware continually acquires radio frequency strength signals (context values) from four reference RFID tags and one RFID tracking tag. The reference tags do not move and their locations are known in advance. Based on these radio frequency strength signals and other environmental context information, the middleware triggers different actions of the applications to compute the estimated location of the tracking tag. One signal from the tracking tag is sufficient for the application to estimate the location of the tracking track within, on average, an uncertainty neighborhood of 1 square meter [12]. It is still an inherent limitation of the existing technology of location-sensing applications to incur a large estimation error.

Table 2 shows 8 estimated locations in the two distinct sensing environments. It gives readers an impression on the accuracy of location estimations. The leftmost column of Table 2 lists the actual locations of the tracking tag, while the other columns list the corresponding estimated locations and the average estimation errors in the environments. All units in this experimentation are in meters. From Table 2, it appears hard to identify failures by simply comparing estimated locations with actual locations. To facilitate experimentation, we use the original version of the location-estimation application as the *golden version*

Table 2. Different estimated locations of *LANDMARC* application.

Actual	Environment 1		Environment 2	
location	Estimated location	Mean error	Estimated location	Mean error
(0.5, 0.5)	(0.825, 1.731)	1.309	(0.407, 0.774)	0.388
(1.5, 0.5)	(1.236, 0.870)	0.473	(1.154, 0.281)	0.481
(0.5, 1.5)	(1.395, 2.598)	1.451	(0.609, 2.786)	1.300
(1.5, 1.5)	(1.009, 1.509)	0.566	(1.400, 2.983)	1.493
(0.5, 2.5)	(1.039, 2.192)	0.663	(0.625, 2.457)	0.267
(1.5, 2.5)	(1.572, 1.364)	1.225	(1.212, 3.525)	1.088
(0.5, 3.5)	(1.220, 3.032)	0.951	(1.587, 3.821)	1.135
(1.5, 3.5)	(0.923, 2.284)	1.355	(0.686, 2.458)	1.336

for reference. We define a test case as a sequence of locations of a tracking tag. Since a tag may move to a location which has been visited before, a test case is thus allowed to contain many copies of the same location.

Ambiance or white noise. As discussed above, the location-estimation application is inherently imprecise when computing its outputs. We treat the imprecision as the *ambiance* or *white noise*, which is the background noise of the environment. We would like to minimize the effect of the ambiance on the checking of our test results. Firstly, we estimate the level of white noise from data sets collected from the two environments. Using a tracking tag at a specific position, we compute the difference between its estimated locations under the respective environments. This measure of the ambiance or white noise provides us with a feel of the estimation error when comparing results from the two environments.

Suppose an actual location is denoted by $\vec{l}_0$ and the estimated locations in Environments E_1 and E_2 are $\vec{l}_1$ and $\vec{l}_2$, respectively. The difference of the location estimates will be $\vec{l}_1 - \vec{l}_2$.

We use the golden version to compute the (golden) ambiance α from the equation

$$\alpha = \frac{1}{n|L|} \sum_{\vec{l} \in L} \sum_{i=0}^{n} |P_G(s(\vec{l})_i, E_1) - P_G(s(\vec{l})_i, E_2)| \tag{15}$$

where (i) L is the set of actual locations of the tracking tag and $|L|$ is the size of L, (ii) n is the number of radio frequency strength signals at each actual location, (iii) $s(\vec{l})_i$ is the i-th signal at actual location $\vec{l}$, (iv) $P_G(s, E)$ is the golden version, which computes an estimated location from the input signal s in Environment E, and (v) E_1 and E_2 are the respective environments. The standard deviation σ corresponding to the mean α is also computed.

Faulty version. In order to investigate the applicability of our technique, we have prepared 21 faulty versions from the golden program that implements the *LANDMARC* location-estimation algorithm. Each faulty version is seeded with a distinct fault. Each seeded fault is of one of the following types: mutation of statement, mutation of operator, missing statement, and missing branch.

Average distance. To compare the same test case under Environments E_1 and E_2, we use the following equation to compute the average distance *avg* between two estimated locations of the same test case:

$$avg = \frac{1}{m|T|} \sum_{\vec{l} \in T} \sum_{i=0}^{m} |P(s(\vec{l})_i, E_1) - P(s(\vec{l})_i, E_2)| \tag{16}$$

where (i) T is a test case and $|T|$ is the path length of T, (ii) m is the number of times for a sequence of locations to re-collect signal data in order to compute the average, and (iii) $P(s,E)$ is a faulty version that computes an estimated location from an input signal s in Environment E. Other variables are used in the same way as in Equation (15).

Test cases. We randomly generate a set of 60 test cases to be applied to the golden version and every faulty version. As discussed above, every test case is a path, which we call a *test path*. In each test path, we set a sufficiently long time interval between any two consecutive input signals, so that the time requirement of relevant situation expressions can be satisfied. Different test paths may have different lengths. We classify the set of test paths into three categories according to their lengths, namely 3 to 9 actual locations, 10 to 19 actual locations, and 20 to 29 actual locations. There are 20 test paths in each category. A test path in each category is generated by randomly selecting s actual locations, where the size s is also determined randomly within the relevant range. To compute the difference between the estimated locations in two environments, for each actual location in a test path, we randomly select 100 radio frequency strength signals out of a total of 3000, which is the entire set of RFID data collected for the particular location. For each test path, we compute the average distance *avg* between the estimated locations in the two environments using Equation (16).

Failure-identification criterion. Test results evaluated according to the value of $|avg - \alpha|$, which is the difference between (i) the average distance of the estimated locations in the two environments and (ii) the white noise level. For a faulty version P, we say that P exposes a failure if and only if $|avg - \alpha| > k\sigma$, where k is a predefined constant.

5.2. *Experimental results and evaluation*

We present the experimental results and discuss our findings in this section. We use the original implementation of *Cabot* and its *LANDMARC* application as the golden version. The experimental configuration consists of the 8 actual locations of the tracking tag under Environments E_1 and E_2 (see Table 2). For each actual location, there are 3000 radio frequency strength signals for location estimation. Based on Equation (15), we compute the white noise to be $\alpha = 1.0397$[k] and the corresponding standard deviation to be $\sigma = 0.6627$.

The classification of test results of all the 21 faulty versions is shown in Table 3. We classify $|avg - \alpha|$ according to different ranges of values of $k\sigma$.

When applying a similarly derived set of 60 test paths to the golden version, as Table 3 shows, we find 53 test paths with $|avg - \alpha|$ in the interval $I_1 = [0, 0.25\sigma]$ and another 7 test paths with $|avg - \alpha|$ in the interval $I_2 = (0.25\sigma, 0.5\sigma]$. For an conservative evaluation, we shall regard a test path to be exposing a failure if and only if the corresponding difference $|avg - \alpha|$ exceeds 0.5σ. Continuing in this way, we compute for each faulty version the percentage of test paths with results falling in one of four other intervals: I_3 (such that $0.5\sigma < |avg - \alpha| \leq 0.75\sigma$), I_4 (such that $0.75\sigma < |avg - \alpha| \leq \sigma$), I_5 (such that $\sigma < |avg - \alpha| \leq 1.25\sigma$), and I_6 (such that $|avg - \alpha| > 1.25\sigma$). The percentages of identified failures

[k] The level of white noise is in line with the average error of one square meter reported in Ni et al.[12]

Table 3. Classification of test results of all 21 faulty versions.

Program under test		Number of test paths with $\lvert avg-\alpha\rvert$ in the range:						% identified	Avg. error in E_1	Avg. error in E_2
		I_1	I_2	I_3	I_4	I_5	I_6			
Golden version		53	7	0	0	0	0	–	0.99	0.92
Faulty version (each with 60 different test paths)	1	55	5	0	0	0	0	0	0.99	0.92
	2	26	24	8	1	1	0	0.167	1.32	***1.42***
	3	0	0	0	0	0	60	1.0	***1.89***	***1.89***
	4	38	21	1	0	0	0	0.017	***1.43***	***1.69***
	5	54	6	0	0	0	0	0	***2.04***	***2.18***
	6	0	0	6	37	17	0	1.0	***1.81***	***1.44***
	7	0	0	0	0	0	60	1.0	***over 3***	***over 3***
	8	0	0	0	0	0	60	1.0	***over 3***	***over 3***
	9	56	4	0	0	0	0	0	0.99	0.92
	10	54	6	0	0	0	0	0	0.99	0.92
	11	51	9	0	0	0	0	0	0.99	0.92
	12	6	33	20	1	0	0	0.35	1.44	1.31
	13	55	5	0	0	0	0	0	***1.79***	***2.20***
	14	42	16	1	0	1	0	0.033	1.01	0.91
	15	53	6	1	0	0	0	0.017	0.99	0.92
	16	0	22	38	0	0	0	0.633	1.14	0.83
	17	42	17	1	0	0	0	0.017	1.00	0.93
	18	50	7	3	0	0	0	0.05	1.02	0.99
	19	34	22	4	0	0	0	0.067	0.92	0.87
	20	0	8	43	9	0	0	0.867	1.31	1.31
	21	0	9	28	14	8	1	0.85	1.32	1.16

where

$I_1:\ \lvert avg-\alpha\rvert \le 0.25\sigma$	$I_4:\ 0.75\sigma < \lvert avg-\alpha\rvert \le \sigma$
$I_2:\ 0.25\sigma < \lvert avg-\alpha\rvert \le 0.5\sigma$	$I_5:\ \sigma < \lvert avg-\alpha\rvert \le 1.25\sigma$
$I_3:\ 0.5\sigma < \lvert avg-\alpha\rvert \le 0.75\sigma$	$I_6:\ \lvert avg-\alpha\rvert > 1.25\sigma$

"% identified" = % of failed test cases identified
(in terms of the fraction of the respective interval I_3, I_4, I_5, or I_6)

are listed in the column "% identified" of Table 3.

We would like to compare our test results with an intuitive approach that checks average estimation error against the golden version. The last two columns of Table 3 show the average estimation error with respect to all the 8 actual positions for the golden version as well as for each faulty version in Environments E_1 and E_2. For the golden version, the average estimation error in E_1 is 0.99 with a standard deviation of 0.45, while the average estimation error in E_2 is 0.92 with a standard deviation of 0.48. Intuitively, an equivalent mutant of the golden version should have an average estimation error within the acceptable level of the benchmark, namely, not exceeding $0.99+0.45=1.44$ in E_1, and not exceeding $0.92+0.48=1.40$ in E_2; otherwise it can be regarded as a failure. The entries of the two columns in ***bold italics*** represent those faulty versions having their average estimation error exceeding the acceptable level in at least one environment.

Our approach identifies 15 out of 21 faulty versions, representing a success rate of 71.4%. A control experiment shows that only 7 faulty versions (representing 33.3%) can be revealed in Environment E_1 using an intuitive approach that reports a failure when the

Table 4. Categorization of faulty versions.

Category	% of failure-revealing test cases	Number of faulty versions	Actual % of failure-revealing test cases		
			Minimum	Average	Maximum
Type 1	0	6	0.0	0.0	0.0
Type 2	> 0 and ≤ 20	7	1.7	5.3	16.7
Type 3	> 20 and < 100	4	35.0	67.5	86.7
Type 4	100	4	100.0	100.0	100.0

estimation error exceeds the acceptance level. Another experiment shows that only 8 faulty versions (representing 38.1%) can be revealed in Environment E_2. Thus, our approach is promising, even though we need to increase the testing effort by running the same test case under distinct environments. On the other hand, the failures of six faulty versions cannot be identified by our technique. A deeper investigation of the seeded faults reveals that the test results of the same test cases under Environment E_1 resemble those of E_2. There is still room for improvement in our approach.

We further categorize faulty versions into 4 types according to the percentages of failure-revealing test cases in the test set. A summary is shown in Table 4. Type 1 represents the faulty versions whose failures cannot be identified. Between zero and 20% of the test cases for faulty version under Type 2 reveal failures. There are 7 faulty versions in this category. A common characteristic is that most of the test paths help identify the failure within $|avg - \alpha| \leq 0.5\sigma$. This may be due to the fact that their faults are minor or else located in program paths that are seldom executed. The remaining two types of faulty versions have relatively high chances for testers to observe their failures. More than 20% but less than 100% of the test cases for faulty versions under Type 3 reveal failures, while all the test cases for faulty versions under Type 4 reveal failures. The faults in these two types are due to significant changes in computation statements or predicates of conditional statements. In particular, the error differences $|avg - \alpha| > 0.5\sigma$ for the faulty versions under Type 4. It is indeed not difficult to observe the failures using whatever testing technique. In short, further investigations will be required to improve our technique with a view to identifying the more subtle failures.

In practice, our testing technique requires a predefined value of the constant k in order to determine whether $|avg - \alpha| > k\sigma$ holds. The selection of an appropriate value of k is non-trivial and may require domain knowledge or recommendations from standardization organizations. Figure 5.2 describes the trends of the distribution of test paths whose results satisfy $|avg - \alpha| > k_i\sigma$ at different levels of k_i. We have selected different values of k_i from the set $\{0, 0.25, 0.5, 0.75, 1.0, 1.25\}$ so that the classification of test results in Table 3 can be utilized.

Figure 5.2 shows six trends of the distributions of test paths. In the overall trend, the vertical axis value of each point is the average number of test paths of all faulty versions that satisfy the predicate $|avg - \alpha| > k\sigma$. For instance, the average number of test paths whose results satisfy $|avg - \alpha| > 0.5\sigma$ is about 20. We observe that if a tester selects a failure-identification criterion such that $k = 0.5$, then about one-third of the test cases will expose the failures. The second trend portrays the distribution of test paths of the golden

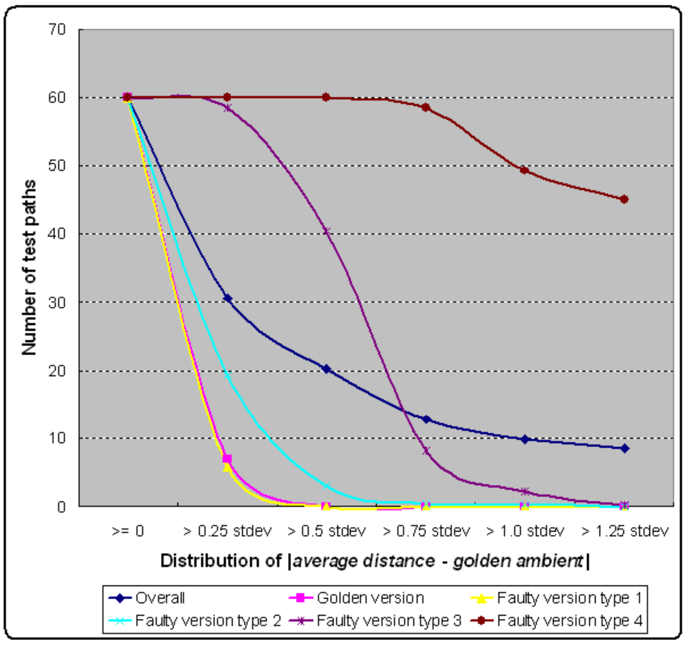


Fig. 4. Distribution of test paths with respect to different levels of $|avg - \alpha|$.

version as a reference. The remaining four trends portray the distribution of test paths of the four categories of faulty versions in Table 4.

We observe from Figure 5.2 that the trend of Type 1 is very close to that of the golden version. Thus, it is difficult to find an appropriate value of k to distinguish them. Compared with the trend of the golden version, the trend of Type 2 faulty versions is not obvious when $k > 0.5$. The trend for Type 3 differs from the trend of the golden version significantly when k is in the interval $(0.5, 0.75)$. It is easy to observe the failures of Type 4.

There are a few potential limitations of the experiments: (i) The testbed is only one of many possible context-aware systems. The characteristics of a location-estimation program may not be relevant to other kinds of application. (ii) We have only studied 21 faulty versions, each having only one fault. They may not represent all types of fault. (iii) We have used only two environments in the experiments. Other environments may produce different results. (vi) We have used only one metamorphic relation in the experiments. Other metamorphic relations may give different results. (v) The prototype middleware may contain faults, so that the outcomes of the golden version may not be correct.

6. Conclusion

Context-sensitive middleware-based software is an emerging kind of ubiquitous computing application. A middleware detects a situation and invokes the appropriate functions of the application under test. As the middleware remains active and the situation may continue to evolve, however, the completion of a test case may not be identified easily. In this paper, we have proposed to use checkpoints as the starting and ending points of a test case. Since the middleware will not activate any function during a checkpoint but may invoke actions in between two such situations, the concept offers a convenient environment for the integration testing of a system.

In our previous work, we demonstrated the ineffectiveness of common white-box testing strategies such as data-flow testing and control-flow testing to detect subtle failures related to situation interfaces. Metamorphic testing with context-decoupled test cases was proposed to reveal failures of context-sensitive middleware-based applications.

In this work, we have further demonstrated the difficulties in revealing the violation of metamorphic context relations involving the execution of multiple context-coupled test cases. To supplement the checking of context relations, we have also proposed to check the relations of execution sequences between checkpoints for multiple test cases. We have illustrated how a subtle failure due to the fault in the example in Section 2.2 can be revealed. In addition, we have reported the results of applying and evaluating our technique to an RFID-based location-sensing system with a context-sensitive middleware.

This paper is a first step toward the integration testing of context-sensitive middleware-based applications. Although the initial results are encouraging, more studies are in order. In particular, we shall develop formal models and systematic procedures for our approach, including the effective identification and selection of checkpoints, the control of the temporal order of adaptive actions during test execution, and the selection of contexts for test oracle evaluation. Based on these models and procedures, we shall investigate the effectiveness of our approach in fault detection, examine the issues of scalability and online testing, develop practical guidelines for our approach, and address the question of automatic checking of metamorphic relations in a context-sensitive middleware-based environment.

Acknowledgments

Special thanks are due to S.C. Cheung and Chang Xu, who have not only provided us with access to their RFID-based system and its raw data, but also given us invaluable insights through their briefing sessions.

References

1. G. D. Abowd and E. D. Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction*, 7 (1): 29–58, 2000.
2. E. W. Axelsen, E. B. Johnsen, and O. Owe. Toward reflective application testing in open environments. In *Proceedings of the Norwegian Informatics Conference* (*NIK 2004*), pages 192–203. Tapir, Trondheim, Norway, 2004.
3. M. Bylund and F. Espinoza. Testing and demonstrating context-aware services with Quake III Arena. *Communications of the ACM*, 45 (1): 29–58, 2002.
4. F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering* (*SE '98*), pages 191–197. ACTA Press, Calgary, Canada, 1998.
5. W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*), pages 241–249. IEEE Computer Society Press, Los Alamitos, California, 2005.
6. T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating

next test cases. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

7. T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering* (*JIISIC 2004*), pages 569–583. Polytechnic University of Madrid, Madrid, Spain, 2004.
8. T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), pages 191–195. ACM Press, New York, 2002.
9. A. Flores, J. C. Augusto, M. Polo, and M. Varea. Towards context-aware testing for semantic interoperability on PvC environments. In *Proceedings of the 2004 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2004*), volume 2, pages 1136–1141. IEEE Computer Society Press, Los Alamitos, California, 2004.
10. A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference* (*COMPSAC 2003*), pages 34–40. IEEE Computer Society Press, Los Alamitos, California, 2003.
11. R. Morla and N. Davies. Evaluating a location-based application: a hybrid test and simulation environment. *Pervasive Computing*, 3 (3): 48–56, 2004.
12. 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.
13. H. J. Nock, G. Iyengar, and C. Neti. Multimodal interfaces that flex, adapt, and persist: multimodal processing by finding common cause. *Communications of the ACM*, 47 (1): 51–56, 2004.
14. E. O'Neill, M. Klepal, D. Lewis, T. O'Donnell, D. O'Sullivan, and D. Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. In *Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities* (*TRIDENTCOM 2005*), pages 60–69. IEEE Computer Society Press, Los Alamitos, California, 2005.
15. I. Satoh. A testing framework for mobile computing software. *IEEE Transactions on Software Engineering*, 29 (12): 1112–1121, 2003.
16. P. Tandler. The beach application model and software framework for synchronous collaboration in ubiquitous computing environments. *Journal of Systems and Software*, 69 (3): 267–296, 2004.
17. P. Tarasewich. Designing mobile commerce applications. *Communications of the ACM*, 46 (12): 57–60, 2003.
18. 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 the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), volume 1, pages 458–465. IEEE Computer Society Press, Los Alamitos, California, 2004.
19. E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25, (4): 465–470, 1982.
20. P. Wu. Iterative metamorphic testing. In *Proceedings of the 29th Annual International Computer Software and Applications Conference* (*COMPSAC 2005*), volume 1, pages 19–24. IEEE Computer Society Press, Los Alamitos, California, 2005.
21. C. Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundation of Software Engineering* (*ESEC 2005/FSE-13*), pages 336–345. ACM Press, New York, 2005.
22. S. S. Yau, D. Huang, H. Gong, and S. Seth. Development and runtime support for situation-aware application software in ubiquitous computing environments. In *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), pages 452–457. IEEE Computer Society Press, Los Alamitos, California, 2004.

23. 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.
24. S. S. Yau, Y. Wang, and F. Karim. Development of situation-aware application software for ubiquitous computing environments. In *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), pages 233–238. IEEE Computer Society Press, Los Alamitos, California, 2002.
25. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit testing coverage and adequacy. *ACM Computing Surveys*, 29, (4): 366–427, 1997.