HKU CS Tech Report TR-2005-05

To appear in *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*),
IEEE Computer Society Press, Los Alamitos, California (2005)

A Metamorphic Approach to Integration Testing of Context-Sensitive Middleware-Based Applications [*] [†]

W. K. Chan [‡]
Hong Kong University of Science and Technology
wkchan@cs.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

Abstract

During the testing of context-sensitive middleware-based software, the middleware identifies the current situation and invokes 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 be easily identified. Moreover, failures appearing in one situation may be superseded by subsequent correct outcomes and may, therefore, be hidden.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 propose to generate test cases that start at a checkpoint and end at another. We identify functional relations that associate different execution sequences of a test case. Based on a metamorphic approach, we check the results of the test case to detect any contravention of such relations. We illustrate our technique with an example that shows how re-hidden failures may be detected.

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

1. Introduction

Context-sensitivity and ad hoc communications [1, 8, 9, 12] are two specific properties of ubiquitous computing applications. The former allows applications to detect, analyze, and react adaptively to changes in attributes, known as the *contexts* [7], 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 [12]. Since the applications operate in a situational and highly dynamic environment, this type of configuration increases the intricacy in software quality assurance. For instance, the ineffectiveness of common testing strategies such as data-flow testing and control-flow testing is exploited and illustrated by examples in [10].

[*] © 2005 IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

[†] 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.

[‡] Part of the research was done when Chan was with The University of Hong Kong.

[§] All correspondence should be addressed to 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.

Besides, the behaviors of the devices can be so volatile that very complicated mathematics is 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*, that is, the mechanism against which testers can check the test outcome and decide whether it is correct. The task will become forbidding if there are a large number of test cases.

We observe that there is a growing amount of research aiming at testing ubiquitous computing applications. Axelsen et al. [2] propose a specification-based approach to test 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 inputs. When the execution sequence of any test input violates the specifications, it detects a failure. Their approach is essentially an execution monitoring approach.

Flores et al. [6] 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 in [6]. 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.

Tse et al. [10] generate multiple context tuples as test cases to check whether the outcomes satisfy isotropic properties of context relations. This idea of applying *metamorphic testing* [4, 5] 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 inputs are then compared. When there is any discrepancy from an expected context relation, an error is revealed.

However, as to be discussed in Section 3, when the contexts change during test case execution, the technique used in [10] to compare resulting contexts may miss to report a failure. Hence, the technique is more applicable if, during the execution of the test case, (i) the contexts remain static *or* (ii) any change in contexts does not affect any situation expression.[1] In this paper, we shall refer to such test cases as *context-decoupled test cases*. Other test cases will be called *context-coupled test cases*. Since situation evolution is a major characteristic of context-sensitive middleware-based applications, context-coupled test cases represent the majority class of test cases.

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

The main contributions of the paper are as follows:

(a) It significantly extends the work of Tse et al. [10] to context-coupled test cases, which allow the updating of contexts during the executions of test cases.

(b) It develops the notion of checkpoints to conduct integration testing, facilitating the checking of test results by the metamorphic testing approach.

(c) It recommends practical guidelines for designing test cases for such applications. For example, the follow-up test cases should activate context-sensitive function(s) via the middleware at chosen checkpoints.

The rest of the paper is organized as follows: Section 2 introduces the work that this paper depends on. 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 discusses our technique through the example of a smart delivery system. Finally, Section 5 concludes the paper.

2. Preliminaries

2.1. Reconfigurable context-sensitive middleware (RCSM)

Reconfigurable Context-Sensitive Middleware (RCSM) [12] is a middleware for the ubiquitous computing environment. It supports a Situation-Aware Interface Definition Language (SA-IDL) [11], for specifying context-sensitive application interfaces. Using an SA-IDL specification [11], 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. Once suitable situational conditions in the SA-spec are detected, the responsible object skeleton will activate appropriate actions.

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, the same action of a situation expression, due to different subgrouping of surrounding devices, may be invoked by the middleware more than once. The "within *x*" clause in a situation expression asserts that the action will be invoked within x seconds after the situation is detected. Similarly, the "frequency = y" clause requires the RCSM to probe the contexts at a rate of y times per second. The "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.

2.2. Smart delivery system: an example

Consider the example of 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. [2] We shall use the word "pallet" to refer collectively to a shelf, pallet, or case in the rest of the paper. The smart deliver 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) Goods that cannot sell 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 for any pallet by 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. The latter will not take replenishment actions if the closest van can provide sufficient quantities. 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.

```
#define   ε      5
RCSMContext Class van extends Base {
  float    q_v;          // the quantity of goods deliverable by a van
  Position    p_v;       // the location of the van in (x, y) coordinates
  float    d;            // square of distance between the van and a pallet
};
RCSMContext 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;}}
RCSMSARule 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 > ε) ∧ (d ⩽ 625));
  PrimitiveSituation     understock
                         ([−3, 0](pallet.q_d − pallet.q_l > ε) ∧ (d ⩽ 265));
                         // Note: 625 is written as 265 by mistake
  ActivateAt overstock{
     [local] void Withdraw( )[within1][priority1]}
  ActivateAt understock{
     [local] void Replenish( )[within1][priority1]}
}
```

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

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 [3] at the pallet site has been short of the desired quantity q_d for more than a tolerance of ε for the last 3 seconds. 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 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

[2] 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".

[3] 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.

for the context variable q_v is computed, it should be detected by the middleware at the pallet site. This paper 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:

$$q_l = q_v + q_p \tag{1}$$

2.3. Metamorphic testing

Metamorphic testing [3–5] is a property-based testing strategy. It recommends that, even if a test case (known as the *original* 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.

Given a function f and its implementation P, a metamorphic relation is a necessary condition over a set of distinct input data $x_1,\ x_2,\ \ldots,\ x_n$ and their corresponding output values $f(x_1),\ f(x_2),\ \ldots,\ f(x_n)$ for multiple executions of the target software f. This relation must be satisfied when we replace f by P; otherwise P will not be a correct implementation of f.

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 $(x_0,\ y_0)$. A sample metamorphic relation is

$$\frac{f(x_1) - y_0}{x_1 - x_0} = \frac{f(x_2) - y_0}{x_2 - x_0}$$

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

Throughout the course of checking of results in metamorphic testing, there is no need to predetermine the expected result for any given input, 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

```
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 qv < MAX {                  s10      if qv > 0 {
  s5          qv = qv + 1;                s11         qv = qv − 1;
        }}                                      }}
  s6    sleep(r/2);                       s12   sleep(r/2);
  }                                       }
```

Figure 2. Implementation of *Replenish*( ) and *Withdraw*( )

results of execution again an oracle, it alleviates the test oracle problem.

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 [4, 5] for the formal definition of metamorphic testing, and [3] for the selection of useful metamorphic relations.

3. Checkpoints in context-sensitive middleware-based applications

3.1. Motivations

Let us first consider the motivations for enhancing the testing technique proposed in our previous work [10]. Given a scenario with one van and one pallet, the tester may generate the following two test cases:

$$\begin{aligned} u_1 &= (s = 1,\ q_d = 100,\ q_l = 50,\ q_p = 0,\ q_v = 7, \\ &\qquad p_p = (1, 1),\ p_v = (0, 0)) \\ u_2 &= (s = 1,\ q_d = 100,\ q_l = 73,\ q_p = 73,\ q_v = 7, \\ &\qquad p_p = (10, 20),\ p_v = (4, 4)) \end{aligned}$$

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 (1). Thus, the following context tuples will result:

$$\begin{aligned} CT_{u_1} &= (\,s=1,\ q_d=100,\ q_l=95,\ q_p=0,\ q_v=95, \\ &\qquad p_p=(1,1),\ p_v=(0,0)\,) \\ CT_{u_2} &= (\,s=1,\ q_d=100,\ q_l=80,\ q_p=73,\ q_v=7, \\ &\qquad p_p=(10,20),\ p_v=(4,4)\,) \end{aligned}$$

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:

First, 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 text 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:

$$\begin{aligned} CT'_{u_2} &= (\,s=1,\ q_d=100,\ q_l=95,\ q_p=73,\ q_v=22, \\ &\qquad p_p=(4,0),\ p_v=(4,16.27)\,) \end{aligned}$$

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*( ) in Section 3.1. 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 stable checkpoint has been reached.

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.

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. First, 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.

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, represent 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 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 reach 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, all 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 [3].

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[4] $(q_{d_1} - \varepsilon \leqslant q_{l_1} \leqslant q_{d_1} + \varepsilon) \wedge d_1 \leqslant 625$ and $(q_{d_2} - \varepsilon \leqslant q_{l_2} \leqslant q_{d_2} + \varepsilon) \wedge d_2 \leqslant 625$.

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 [10].

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 an original 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', 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', the number of invocations of the *Withdraw*( ) function for t' is expected to more be than that of t.

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

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

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

Updated contexts for t_2

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. In theory, therefore, testers may not be able to distinguish Case (3) from Case (2). In practice, testers may assume that the software will not terminate if 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, however, we shall restrict ourselves to only Case (2) in this paper.

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 in 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*, using context-coupled test cases.

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. 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 *original* test case.

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

Step (1): Apply the initial context CT_t to the one pallet and one van configuration. Update the derived context d to 256.[5] According to Equation (1), 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 of the possible ways 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 ε = 5, and since d is not more than 265, the situation *understock* is detected and,

[5] 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.

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.

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 decremented 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[6] of $\vec{\Theta}_{t_1}$. 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 original 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 an original 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. First, 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,

[6] 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 $\vec{\Theta}_{t_1}$.

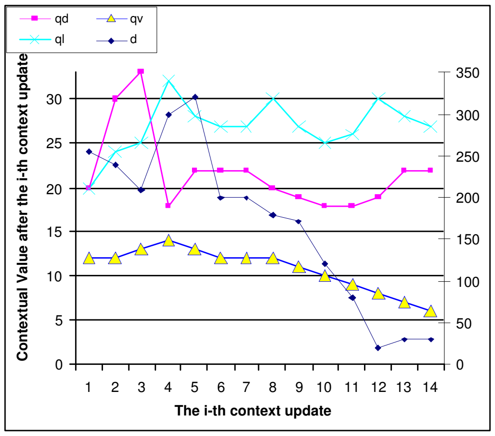

Figure 3. Context trends for test case t_2.

$p_2 = (s' = 1,\ q'_d = 100,\ q'_n = 73,\ q'_o = 73,\ p'_v = (10, 20))$. 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:

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

We first verify the results at checkpoints. Since the metamorphic relation MR_1 is applicable, testers may apply

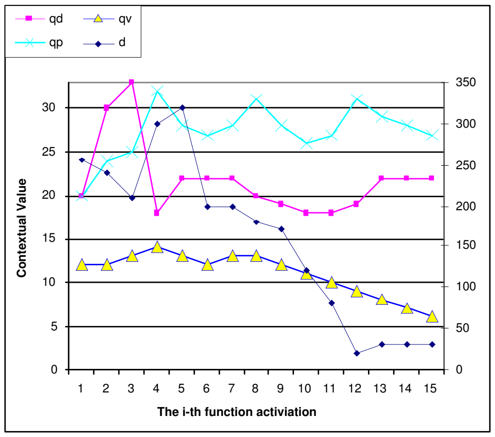

Figure 4. Expected context trends for test case t_2.

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. Figure 3 shows graphically the trends of the context variables for test case t_2. 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 4 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.

5. 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 conducting the integration testing of the functions 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.

We have significantly extended our previous work. This paper is a first step toward the integration testing of context-sensitive middleware-based applications. Although the initial results are encouraging, further investigations and experimentations are in order. In particular, we shall investigate the effectiveness of our approach in fault detection, examine the issues of scalability and online testing, develop formal procedures and practical guidelines for our approach, and address the question of automatic checking of metamorphic relations in a context-sensitive middleware-based environment.

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] 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.

[4] 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.

[5] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45 (1): 1–9, 2003.

[6] 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.

[7] 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.

[8] 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.

[9] P. Tarasewich. Designing mobile commerce applications. *Communications of the ACM*, 46 (12): 57–60, 2003.

[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 the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), volume 1, pages 458–465. IEEE Computer Society Press, Los Alamitos, California, 2004.

[11] 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.

[12] 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.