Testing Context-Sensitive Middleware-Based Software Applications *

T. H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
thtse@hku.hk

Stephen S. Yau
Computer Science & Engineering Department
Arizona State University
Tempe, AZ 85287, USA
yau@asu.edu

W. K. Chan
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
wkchan@cs.hku.hk

Heng Lu
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
hlu@cs.hku.hk

T. Y. Chen
School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
tchen@it.swin.edu.au

Abstract

Context-sensitive middleware-based software is an emerging kind of ubiquitous computing application. The components of such software communicate proactively among themselves according to the situational attributes of their environments, known as the "contexts". The actual process of accessing and updating the contexts lies with the middleware. The latter invokes the relevant local and remote operations whenever any context inscribed in the situation-aware interface is satisfied. Since the applications operate in a highly dynamic environment, the testing of context-sensitive software is challenging.

Metamorphic testing is a property-based testing strategy. It recommends that, even if a test case does not reveal any failure, follow-up test cases should be further constructed from the original to check whether the software satisfies some necessary conditions of the problem to be implemented. This paper proposes to use isotropic properties of contexts as metamorphic relations for testing context-sensitive software. For instance, distinct points on the same isotropic curve of contexts would entail comparable responses by the components. This notion of testing context relations is novel, robust, and intuitive to users.

Keywords: *Property-based testing, RCSM, middleware-based application, metamorphic testing*

1. Introduction

Context-sensitive middleware-based software is an emerging kind of computing application following up on the concept of ubiquitous computing, or computing everywhere. The *context* of an entity is any information characterizing its environmental situation [1]. The components of context-sensitive software communicate proactively among themselves according to the contexts. Various projects, such as [2, 3, 4, 5, 6, 7, 8, 9], employ a context-sensitive middleware to assess the environment so that the low-level recognition process can be hidden from the users' applications. Pilot applications such as [10, 11, 12, 13] have been reported in the literature. Since applications must operate in a highly dynamic and situated environment, this type of configuration increases the intricacy in software quality assurance. To our best knowledge, there is *no* software testing technique addressing context-sensitive middleware-based applications, although testing is the major means to assure

* This work is supported in part by a grant of the Research Grants Council of Hong Kong, a grant of the Croucher Foundation, a grant of The University of Hong Kong, and an Australian Research Council Discovery Grant (Project No. DP 0345147).

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

their quality. Finding effective software testing techniques for such applications in a specification-based or program-based setting is an open and challenging problem.

In conventional approaches in software testing, the behavior of an application is assumed to be included inside the implemented program. In context-sensitive middleware-based applications, on the other hand, the middleware may repeatedly invoke certain software components according to the interface contexts, until the triggering conditions inscribed in the middleware are no longer satisfied. Hence, part of the application behavior can be determined by a triggering condition or a stopping criterion specified in the middleware rather than based on the source code of the application. This blurred boundary poses new challenges to software testers. Even for unit testing, it is not sufficient to consider only the source code of the application (such as when constructing test cases for *all-du* coverage [14] in white-box testing), or to use the situational conditions registered in the middleware as activation conditions (in the sense of pre-conditions in model-based languages such as Z [15]). Furthermore, it is a formidable task to work out a precise test oracle and to test the application against it.

Metamorphic testing [16, 17, 18] is a property-based testing strategy. It recommends that, even if a test case does not reveal any failure, follow-up test cases should be further constructed from the original to check whether the software satisfies some necessary conditions of the problem to be implemented. It can reveal functional errors without the need to rely on test oracles. Consider a (metamorphic) relation for more than one input-output pair, such as $(x_1 - x_0)^2 + (f(x_1) - y_0)^2 = r^2 = (x_2 - x_0)^2 + (f(x_2) - y_0)^2 \ \wedge \ x_2 = 2x_0 - x_1$. When all but one input-output pairs are known, such as $x_0 = 2$, $y_0 = 2$, $x_1 = 1$, and $f(x_1) = 1$, we can compute the *next* input, say $x_2 = 3$. Furthermore, we can determine whether this input-output pair, say $(x_2,\ y_2) = (3,\ f(3))$, violates the metamorphic relation. Throughout the course of checking of results, there is no need to pre-determine the expected value for any particular input, such as whether $f(3)$ should or should not be 1.

It is obvious from this example that a metamorphic relation is not the same as the specification for an application. The former does not define the expected outcomes in an explicit form. It facilitates software testing in cases where it is difficult to determine the test oracle precisely. Passing every test case in metamorphic testing does not warrant the correctness of an application. On the other hand, this is the limitation of all testing methods.

The rest of the paper is organized as follows: First, Section 2 describes the infrastructure of context-sensitive middleware-based applications, which is the main topic of interest of the paper. It will pay special attention to context-sensitive interfaces from the viewpoint of software testing. This is illustrated by a smart streetlight application described in Section 3. Section 4 discusses the difficulties of testing such systems. Sections 5 and 6 review metamorphic testing and demonstrate how this can be applied to reveal the failures in the smart streetlight example. Finally, Section 7 concludes the paper.

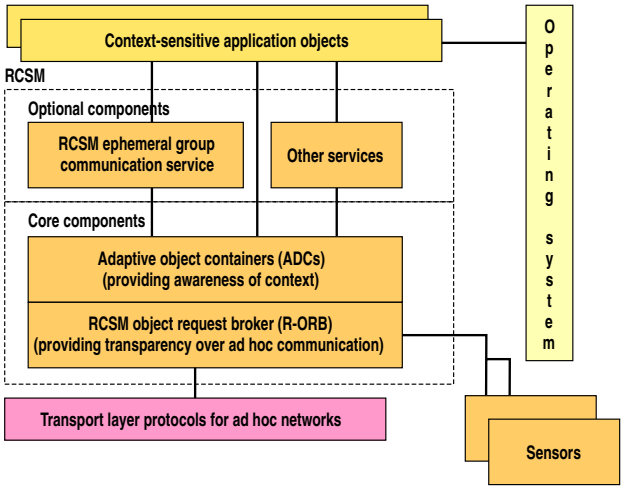

Figure 1. The device-centric architecture of RCSM (from [8]).

2. Reconfigurable Context-Sensitive Middleware (RCSM)

2.1. Architecture

According to [19, 20, 21, 22], an application in ubiquitous computing environment [23] exhibits two properties. First, an application is *context-sensitive* when it adapts its behavior by using information from its surrounding environment, usually known as the contexts. Secondly, such applications communicate frequently and proactively with other devices in an ad hoc network.

Reconfigurable Context-Sensitive Middleware (RCSM) [8] is a middleware for the ubiquitous computing environment. It responds to these two properties by providing a context-sensitive interface. It allows applications to free themselves from the detection of contexts and concentrate on context-independent actions. More specifically, RCSM regards each context-sensitive application as an object and provides the latter with a custom-made adaptive object container (ADC) generated according to the RCSM-specific interface definition specification [24].

Figure 1, taken from [8], sketches the architecture of RCSM in a typical device. During runtime, each adaptive object container will register its contextual requirements

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

to the middleware and periodically collect raw contextual data from the underlying system. Once suitable situational conditions are detected, the responsible adaptive object container will activate appropriate actions. The discovery of devices, the communication model among devices, and the detection of any specific situations are transparent to applications. More detailed explanations can be found in [8, 25].

2.2. Situation-Aware Interface Definition Language

Situation-Aware Interface Definition Language (SA-IDL) [26] describes formally the situations to be detected and the corresponding actions to be passed on to the applications.

A *time stamp*, denoted by t, carries the ordinary meaning of time as represented in most systems. A *context variable*, denoted in general by c_i, specifies an attribute of a specific context. A *context tuple*, or simply a *context*, is a tuple $\langle t,\ c_1,\ c_2,\ \ldots,\ c_n\rangle$ of context variables. For instance, suppose that $GPS_{position} = (x,\ y,\ z)$ is a context variable representing the position of a moving trolley. A context describing its position can be expressed as $\langle t,\ GPS_{position}\rangle$. A *derived context* is a mathematical function of contexts that describes how contexts vary with time. An *action tuple* is a tuple $\langle t,\ a_1,\ a_2,\ \ldots,\ a_n\rangle$ of actions in response to a specific context. In the rest of the paper, we shall assume the existence of the time component without an explicit reference in our examples.

A *situation expression* indicates how contexts and actions vary over time. A valid situation expression includes the following components in sequence: (*i*) a universal or existential quantifier, (*ii*) the variable t within a time range, and (*iii*) a list of comparisons among actions, contexts, and values. A situation expression can also be composed with other situation expressions to form a new situation expression using the logical operators "*and*", "*or*", or "*not*". In this paper, conditions in a situation expression are also referred as situational conditions. Finally, a responding action must be annotated with one of the following tags: [*incoming*], [*outgoing*], [*local*], or [*clientserver*]. It should also be associated with the context using a tag [*activate at context x*], where x is a context variable or expression. More detailed elaborations on the specification format and how it deals with real-time and quality-of-service (QoS) issues can be found in [25].

3. Smart Streetlight System: an Application Scenario

3.1. Description

Consider an example of a system of smart streetlights that collaborate to illuminate a city zone. The system includes two features. (*i*) As far as possible, every visitor can personalize their favorite level of illumination irrespectively of their location within the zone. (*ii*) At the same time, the system maximizes energy savings by dimming unnecessary streetlights.

When there is no visitor nearby, a streetlight will turn itself off. When a visitor walks toward a particular streetlight, the light detects the visitor and brightens itself. Other surrounding streetlights may dim if the closest light has provided sufficient illumination. The other streetlights may not dim, however, if there are other visitors requiring illumination. Because of the interference from other light sources and the presence of other visitors nearby, the resulting illumination for the visitor may differ from their favorite level. Finally, the system assumes that the effective distance for any streetlight to serve a visitor is at most 5 meters.

3.2. Sample Programs

Figure 2 [1] shows a sample situation-aware interface definition specification for a lighting device [2]. In particular, the situation *low_illumination* targets to represent that, when the visitor is inside the effectively illuminated region at time stamp t of the received context, the current illumination l_n at the visitor site is short of the favorite illumination l_f for more than a tolerance of ε in the past 3 seconds. When this is the case, the application needs to power up its lighting device. This is accomplished by invoking the local function *PowerUp*( ). A situation *high_illumination* is similarly defined.

Nevertheless, there is an error in the SA-IDL specification of lighting devices as shown in Figure 2. In the situation expression *low_illumination*, the variable d is mistaken to mean "distance" rather than its square. It defines the situation expression to be detected when the value of the variable d is no more than 5 meters, which would be conceptually correct if the variable d were indeed a measure of the distance. The correct comparison should check its value against "25", and the correct situation expression for the situation *low_illumination* should be as follows:

$$\begin{array}{lll} \text{Situation} & \textit{low\_illumination} & \\ & \text{ForAny} \quad (t > T-3) & \\ & \quad (\boldsymbol{d} \leqslant \mathbf{25}) \ \wedge \ (l_f - l_n > \varepsilon) & \end{array} \tag{1}$$

[1] In practice, a derived context should be used in situation expressions to compute the differences between the variables l_f and l_n. For the ease of presentation in the paper, the specification in the figure uses a simplified notation that merges the derived context definitions for $(l_n - l_f)$ and $(l_f - l_n)$ into their corresponding situation expressions.

[2] To simplify our discussion in this paper, every context is placed in the same context tuple in the SA-IDL specification.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

```
#define ε 0.1

ContextTuple light_context{
  Time t;          // time stamp

                   // remote contexts:
  int s;           // no. of surrounding streetlights of a visitor
  float l_f;       // the favorite illumination
                   // (radiance) of the visitor
  float l_n;       // the illumination (radiance)
                   // at the visitor site
  Position p_v;    // visitor's (x, y) position

                   // local contexts:
  float l_0;       // the illumination (radiant intensity)
                   // emitted from the streetlight
  Position p_l;    // (x, y) position of the streetlight
};

interface smart_light{
  Derived   d  (p_l.x − p_v.x)^2 + (p_l.y − p_v.y)^2)

  Situation high_illumination
            (ForAny t > T − 3)  (d ⩽ 25) ∧ (l_n − l_f > ε)
  Situation low_illumination
            (ForAny t > T − 3)  (d ⩽ 5) ∧ (l_f − l_n > ε)
            // Note: (d ⩽ 25) is mistaken as (d ⩽ 5)
    [local][activate at high_illumination] void PowerDown( );
    [local][activate at low_illumination] void PowerUp( );
  };
```

Figure 2. A simplified SA-IDL for the smart lighting device.

The concept behind the function *PowerUp*( ) is as follows: In the smart streetlight system, each visitor will be surrounded by a number, say s, of streetlights that can communicate with the visitor. Suppose that the lighting is optimal at the moment and a visitor increases their favorite illumination so that exactly one surrounding streetlight can optimally meet its new requirement. Since there are s streetlights detecting the change, there are s devices ready to activate their *PowerUp*( ) functions. Obviously, these streetlights must be collaborative; otherwise an uncontrolled effect will result in too much or too little light for the visitor. In each device for a streetlight, the implementation *PowerUp*( ) makes use of the contextual data s that represents the number of lights surrounding the visitor. The function computes the probability $\frac{1}{s}$ that it needs to increase the power supplied, and then casts a die. Since there are s surrounding streetlights and each has a probability of $\frac{1}{s}$ to brighten itself, there will be, on average, one streetlight to serve the visitor. When

```
void PowerUp( ){
 s1  int r;
 s2  r = rand( ) % s;
     // randomize the action
 s3  if r == 0 {
 s4    if l_0 < MAX {
 s5      l_0 = l_0 + 1;
     }}
 s6  sleep(r/2);
 }
```

Figure 3. The implementation of *PowerUp*( ).

none of the streetlights chooses to light up, the situation *low_illumination* will remain active. Further running of the *PowerUp*( ) function will be required.

On the other hand, when there is more than one streetlight lighting up, the situation *high_illumination* at all the surrounding streetlights will be activated accordingly. The function *PowerDown*( ) will be run to dim the corresponding lights non-deterministically.

To complete an overall adjustment, an individual streetlight may or may not activate the functions *PowerUp*( ) and *PowerDown*( ). In general, each of these functions will make small adjustments to the power supply and, hence, the corresponding middleware is required to invoke the functions a number of times to achieve the required illumination. Consequently, the overall illumination at the visitor site will *oscillate*, sometimes higher than the expected and sometimes lower, and will eventually reach an optimal value.

Figure 3 shows a correct implementation of the function *PowerUp*( )[3]. Once a new value for the context variable l_0 is computed, it should be detected by the middleware at the visitor site. This paper assumes that there is a correct test stub for the function *ComputeRadiance*( ) in the visitor device to take the values of l_0 from all the surrounding streetlights and to compute a corresponding new value for the context variable l_n. The theoretical formula to compute the variable l_n is defined as follows[4], although tolerances such as $|l_n - l_f| < \varepsilon$ may need to be added in real-life practice.

$$l_n = \sum_{i=1}^{s} \frac{l_0^{(i)}}{d^{(i)}}$$

where $l_0^{(i)}$ and $d^{(i)}$ denote the context variable l_0 and

[3] In practice, the value of the context variable l_0 may be passed to a control system to regulate the power supply after statement s_5.

[4] According to the principles of optics, the value of illumination emitted by the streetlight is expressed in terms of radiant intensity, denoted by l_0, and the value of illumination estimated at the visitor site is expressed in terms of radiance, denoted by l_n. They obey the formula $l_n = k\frac{l_0}{d^2}$, where d is the distance between the streetlight and the visitor, and k is a constant.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

the derived context variable d, respectively, from the ith surrounding streetlight. In particular, for a configuration with only one visitor and one streetlight, the formula can be simplified to:

$$l_n = \frac{l_0}{d} \tag{2}$$

4. Challenges in Testing Context-Sensitive Middleware-Based Software

Most context-sensitive middleware-based systems emphasize that context detections, situation detections, and invocations of corresponding functions are the duties of the middleware. However, the active nature of the middleware and the emphasis of clear segregation of duties pose new challenges to testers.

An SA-IDL specification is a readily available formal definition of the interface between an application and the middleware in an RCSM framework. An error in the specification, which will automatically be translated into problematic code by the SA-IDL compiler, may be difficult to detect for a number of reasons. First, there are many challenges in testing the prohibitive number of possible relationships between the application and the middleware, as discussed later in this section. Secondly, an error-free SA-IDL specification is only an idealistic target and is not known to the designer. There is, therefore, no oracle to validate the correctness of the functions defined in an SA-IDL specification. Thirdly, one may also like to adopt a formal specification to specify the functions of the application, thereby easing the development and testing process. However, since they involve probabilistic operations, their specifications must either be probabilistic or non-deterministic. Take *PowerUp*( ) as an example. An informal non-deterministic description may take the form: "As long as the context variable l_0 is less than *MAX*, *PowerUp*( ) may or may not increment l_0 by 1." The integration of a formal non-deterministic specification with an SA-IDL specification to help detect, for instance, the missing-situation error in Figure 2, is subject to extensive further research.

For unit tests in conventional software testing, a function such as *PowerUp*( ) plus the associated situations can be treated in at least 3 different levels with a middleware that can synchronize contexts:

*L*1: Treat *PowerUp*( ) as a function under test in a conventional application.

*L*2: On top of level *L*1, consider situational conditions as constraints imposed on the input domain. In other words, every test input has to satisfy these constraints; otherwise the middleware will not activate the program under test accordingly.

*L*3: In addition to level *L*2, consider the middleware to be autonomous in function invocations.

They also represent different degrees of challenges in testing.

4.1. Level *L*1

Suppose there is a fault in statement s_5 of the *PowerUp*( ) implementation such that it updates the variable l_n instead of l_0:

$$\overline{s_5}: \qquad l_n = l_0 + 1;$$

Suppose a tester follows this faulty implementation of *PowerUp*( ) to partition the set of legitimate execution paths according to the path analysis strategy [27]. Three distinct execution paths will result. Consider the path $\langle s_1, s_2, s_3, s_4, \overline{s_5}, s_6\rangle$ that passes through all the statements. Suppose

$$\langle s = 1,\ l_f = 2,\ l_n = 5,\ l_0 = 7,\ p_v = (2,\ 3),\ p_l = (0,\ 0)\rangle$$

is the initial value of the context tuple for the streetlight. An execution of this path will modify the context variable l_n from 5 to 8 because of the faulty statement $\overline{s_5}$. A correct computation should not amend the value of l_n, but should adjust l_0 to 8 instead.

There are at least two challenges in this case:

(*a*) The context variables are detected and probed in real time by the underlying middleware. Since l_n is a remote context variable, a new value for this context variable may supersede the computation error at any time, so that the fault may not be detected. For instance, according to formula (2), l_n at the visitor site will be updated to $\frac{7}{13}$ when l_0 is 7, and to $\frac{8}{13}$ when l_0 is 8. This value will then synchronize with the local context tuple at the lighting device [5].

One can argue that this is the usual problem of a race condition (that is, having concurrent operations that conflict with one another) as in conventional concurrent applications. On the other hand, it is also a desirable property of the middleware to refresh context variables in order to provide transparency to the application. However, this may result in an undetected fault.

(*b*) Suppose the middleware were not included in the unit testing. In this case, testers would be modifying the behavior of the application. We would not be able to draw a conclusion whether it would be an error for the test result of l_0 to remain as 7, unless there was a detailed specification for testers to compute the behaviors when the effects of the middleware were totally diminished.

[5] During unit testing, testers tend to implement test stubs at the streetlight site to simulate the actual implementation at the visitor site.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

4.2. Level *L*2

The second level of unit testing is to take the situational conditions into account. By treating the situational conditions in SA-IDL specification as constraints to define the input domain, testers may apply the domain-testing strategy [28]. Testers may partition the input domain into two subdomains according to the situational conditions *low_illumination* and *high_illumination*.

However, as situational conditions play an active role in the behaviors of the application, a variant in the situation expression may cause the function (such as *PowerUp*( )) to be activated improperly or not to be activated properly. For instance, the situation expression in equation (1) of Section 3 shows that the function *PowerUp*( ) should be activated when the streetlight is at position $(0,\ 0)$ and the visitor is at position $(3,\ 3)$. (That is, the derived context variable d in the SA-IDL specification will be evaluated to 18, which is less than 25. Hence, the middleware will trigger the function *PowerUp*( ) according to the situation expression in equation (1) [6].)

4.3. Level *L*3

The third level is to take into account the active nature of the middleware. Hence, a unit under test is the integrated module of a triggered function (such as *PowerUp*( )) and the related triggering situation expressions (such as *low_illumination*). As contexts are partially controlled by middleware, (intermediate) values and validities of context variables may change in unforeseeable combinations during a series of automatic triggering. The number of potential combinations is usually formidably large. It poses a combinatory explosion problem to testers.

Furthermore, as the middleware is proactive, it may trigger application functions now and then. Thus, it will not be easy to determine precisely the (hard) termination of a computation for a particular input. Consider, for example, the faulty statement $\overline{s_5}$ in *PowerUp*( ) and the initial context tuple discussed in level *L*1 above. Since the illumination l_n takes a value of $\frac{7}{13}$ at the visitor site, the situation *low_illumination* will be invoked, causing the local copy of l_n to be erroneously altered to 8. The latter will be detected by the middleware and, hence, the computation will not terminate. On the other hand, even for the correct implementation of the application, since the implementation *PowerUp*( ) is non-deterministic, it may take an indefinite number of invocations before the situation *low_illumination* can be satisfied to terminate the execution of a test case.

[6] We observe that a context tuple meeting the constraints of situation expression *low_illumination* in Figure 2 will also satisfy this (correct) situation expression shown in equation (1). Thus, any test case that can cause the middleware to trigger the function *PowerUp*( ) will *not* reveal this problem.

4.4. Inadequacies of Data-Flow Testing and Coverage Testing

The data-flow testing strategy, code coverage strategy (such as path coverage), and predicate-based testing strategy are three of the most popular kinds of strategy in code-based testing techniques. They aim at generating test cases so that different parts of the program under test will execute. As discussed in Section 4.2, the fault in the program under test lies in the context-sensitive interface that excludes certain situations to be detected by the middleware. Thus, by generating test cases according to the structure of *PowerUp*( ) and its relationships to context variables, and at the same time fulfilling the SA-IDL constraints to activate the function *PowerUp*( ), one can never discover the missing situations. For the unit under test (*PowerUp*( ) with situation *low_illumination*), for example, the following test set fulfills the *all-branch-coverage* criteria for code coverage strategy, the *all-du* coverage criteria for data-flow testing strategy, and the *all-predicate-use* criteria for predicate-based testing strategy at the same time.

$$\left\{ \begin{array}{l} \langle s=1,\ l_f=10,\ l_n=5,\ l_0=7,\ p_v=(1,\ 1),\ p_l=(0,\ 0)\rangle, \\ \langle s=1,\ l_f=10,\ l_n=5,\ l_0=MAX,\ p_v=(4,\ 0),\ p_l=(0,\ 0)\rangle \end{array} \right\}$$

Nevertheless, *no* failure can be revealed.

On the other hand, by sufficiently modifying these testing techniques so that they can (creatively) produce test cases to cover these supposedly "infeasible" situations seem to violate their underlying philosophy and may produce a great deal of illegitimate test cases. For instance, the effective serving distance of a smart streetlight is at most 5 meters. The data type "float" for the variable d includes mostly numbers larger than 5.

5. Metamorphic Testing

A metamorphic relation [16, 17, 18] is an existing or expected relation over a set of distinct input data and their corresponding output values for multiple executions of the target program. Metamorphic testing is a property-based testing strategy based on such relations. If a group of test cases and their corresponding outputs do not satisfy a specific metamorphic relation, then the program under test must contain a fault. Various studies on metamorphic testing have been carried out. Reference [30] tests the implementation of partial differential equations. References [17, 18] investigate the integration of metamorphic testing with fault-based testing and global symbolic evaluation. Reference [31] develops an automated framework.

For context-sensitive middleware-based applications, contexts can be parts of the inputs and parts of the

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

outputs of a feature[7] at the same time. If metamorphic relations among contexts can be established, testers can apply metamorphic testing to such applications. There are a few advantages. First, testers can alleviate the problem of blurred boundary between the context-sensitive function and the situation-aware interface, as raised in Section 1. Secondly, testers can check properties of the software that are independent of situational conditions. Thirdly, it models the feature under test as a black box and, hence, the internal explosion of the legitimate combinations of context tuples is encapsulated.

A metamorphic relation can be formally described as follows: Suppose that f is an implemented function under test. Given a relation r over n distinct inputs, $x_1, x_2, \ldots, x_n$, the corresponding n computation outputs $f(x_1), f(x_2), \ldots, f(x_n)$ must induce a necessary property r_f. The corresponding formula can be expressed as

$$r(x_1,\ x_2,\ \ldots,\ x_n) \Rightarrow\ r_f(f(x_1),\ f(x_2),\ \ldots,\ f(x_n))$$

In other words, a metamorphic relation of f over n inputs and n outputs can be defined as follows:

$$MR_f\ =\ \{(x_1,\ x_2,\ \ldots,\ x_n,\ f(x_1),\ f(x_2),\ ...,\ f(x_n)) \mid r(x_1,\ x_2,\ \ldots,\ x_n) \Rightarrow\ r_f(f(x_1),\ f(x_2),\ \ldots,\ f(x_n))\}$$

6. Testing Context-Sensitive Properties

In this section, we apply the notion of metamorphic testing to the smart streetlight application described in Section 3 in a configuration involving a visitor and a streetlight. As described in Section 3, a feature of this application is that it can provide similar levels of illumination to a visitor at different locations in the streetlight zone. This intuitive isotropic service agreement is expressed as a situational condition $l_f - l_n > \varepsilon$. Hence, whenever the visitor is within the effective servicing area of the streetlight, a correct implementation of SA-IDL and *PowerUp*( ) under a test stub for *ComputeRadiance*( ) using formula (2), should compute l_n to a value within the specified tolerance limit. In this case, the maximum tolerance between two distinct values of l_n can at most be 2ε. Consequently, we have the following metamorphic relation for the unit testing of the function *PowerUp*( ):

$$MR_{PowerUp}\ =\ \{(p_1,\ p_2,\ l_{n_1},\ l_{n_2},\ l_{f_1},\ l_{f_2},\ p_0,\ r_{eff}) \mid r^2(p_1,\ p_0) \leqslant r_{eff}^2 \wedge\ r^2(p_2,\ p_0) \leqslant r_{eff}^2 \wedge\ l_{f_1} = l_{f_2} \Rightarrow\ l_{n_1} \approx l_{n_2}\}$$

where

(*i*) r_{eff} is the radius of the effective illumination region of a streetlight,

(*ii*) $r^2(p_i,\ p_0)$ is a function to return the square of the distance between the streetlight at position p_0 and the visitor at position p_i,

(*iii*) l_{f_1} and l_{f_2} are the favorite illuminations of the visitor at positions p_1 and p_2, respectively, and

(*iv*) the symbol $\approx$ denotes that the two values are approximately equal within an application-specific tolerance limit of 2ε.

Application designers, users, or experienced testers can obviously propose such context-sensitive properties for testing. The metamorphic relation above for the smart streetlight application, for example, can be intuitively produced from informal requirements descriptions together with an understanding on the SA-IDL specification.

Testers may generate lists of context variables as test cases for the metamorphic relation $MR_{PowerUp}$. For instance, both of the following two lists t_1 and t_2 have visitor positions inside the effective illumination region. Furthermore, the testing of non-determinism due to the random function *rand*( ) can be achieved using such methods as the forced deterministic testing approach [32, 33].

$$t_1\ =\ \langle s = 1,\ l_f = 10,\ l_n = 5,\ l_0 = 7,\ p_v = (1,\ 1),\ p_l = (0,\ 0)\rangle$$
$$t_2\ =\ \langle s = 1,\ l_f = 10,\ l_n = 5,\ l_0 = 7,\ p_v = (4,\ 0),\ p_l = (4,\ 4)\rangle$$

(*a*) For the test case t_1, 2 will be assigned to the derived context variable d. Hence, the predicate $d \leqslant 5$ will be evaluated to true. Moreover, as the initial value of l_n ($= 5$) is also smaller than that of l_f ($= 10$), even if a tolerance threshold ε is taken into account, the situation *low_illumination* should be detected by the middleware. The middleware, thus, invokes the function *PowerUp*( ).

After the first round of execution of *PowerUp*( ), l_0 is increased from 7 to 8. This change in value for the variable l_0 will be passed to the test stub for *ComputeRadiance*( ), which computes a new value $\frac{8}{2} = 4$ for the variable l_n. The middleware will still detect this value of l_n as satisfying the situation *low_illumination*. Additional invocations of the function *PowerUp*( ) will be made. The iterative process will be repeated 12 more times, so that l_0 is gradually increased to 20. The test stub for *ComputeRadiance*( ) computes the latest l_n to be 10.

[7] A *feature* includes both the context-sensitive function under test and its corresponding SA-IDL.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

The situation *low_illumination* will no longer be satisfied. The context tuple will be

$$CT_{t_1} = \langle s = 1,\ l_f = 10,\ l_n = 10,\ l_0 = 20,\ p_v = (1,\ 1),\ p_l = (0,\ 0)\rangle$$

and will remain unchanged afterwards.

(*b*) On the other hand, for the test case t_2, the variable d will be computed to give 16, which is larger than 5. Thus, the predicate $d \leqslant 5$ will be evaluated to false, so that the situation *low_illumination* will never be activated. Since the initial value of l_0 is 7 and stays constant, l_n is updated to $\frac{7}{16}$, or 0.4375. It will preserve this value afterwards. The eventual context tuple of t_2 will be

$$CT_{t_2} = \langle s = 1,\ l_f = 10,\ l_n = 0.4375,\ l_0 = 7,\ p_v = (4,\ 0),\ p_l = (4,\ 4)\rangle$$

By the metamorphic relation $MR_{PowerUp}$, the difference between l_n in CT_{t_1} and that in CT_{t_2} exceeds the tolerance limit 2ε ($= 0.2$). Consequently, since the *eventual*[8] values of the context variable l_n for the two test cases are not always the same within certain tolerance limits over a period of time, the relation $MR_{PowerUp}$ is violated. In other words, it reveals a failure in the implementation under test.

7. Conclusion

Ubiquitous computing is a notion of computing everywhere. One of the emerging approaches is to develop context-sensitive middleware that facilitates application-transparent communications in an ad hoc network. This paper examines the active nature of the middleware based on RCSM. We note that there are challenges for testers to test applications atop such middleware. They include (*i*) race conditions in context tuples between the middleware layer and the application layer, (*ii*) potential non-testable nature of situation expressions, and (*iii*) combinatory explosion of unforeseeable combinations of intermediate contexts to trigger subsequent context-sensitive functions.

This paper proposes to investigate into the metamorphic relations of the context tuples so that the program in the middleware under test can be modeled as a black box. In this way, race conditions and state explosions of intermediate contexts can be encapsulated. We also propose that, owing to the non-testable nature of situation expressions, metamorphic relations can be chosen to be independent of situation expressions. Hence, such a metamorphic relation is black-box and situation-independent, thus providing a robust testing platform for complex context-sensitive software systems.

[8] In the sense that it is sufficiently long from the real-time perspective to affect outputs.

The paper also describes a smart streetlight example. It uses the service level agreement as an informal specification to formulate a metamorphic relation. Based on the relation, we discuss a way to detect missing-situation errors in an implementation of the power-up feature of the example system.

In summary, the application of metamorphic testing to context-sensitive middleware-based software systems is novel, robust, and intuitive to testers.

8. Acknowledgements

The authors would like to thank the anonymous reviewers for their very encouraging evaluations of the paper.

References

[1] H. J. Nock, G. Iyengar, and C. Neti. Multimodal processing by finding common cause. *Communications of the ACM*, 47 (1): 51–56, 2004.

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

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

[4] M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA applications in a mobile environment. In *Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 36–47. ACM Press, New York, 1999.

[5] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. Xmiddle: a data-sharing middleware for mobile computing. *Wireless Personal Communications*, 21 (1): 77–103, 2002.

[6] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering* (*ICSE 2001*), pages 368–377. IEEE Computer Society Press, Los Alamitos, California, 1999.

[7] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces: the next wave. *IBM Systems Journal*, 37 (3): 454–474, 1998.

[8] 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.

[9] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12 (3): 317–370, 2003.

[10] G. Cabri, L. Leonardi, and F. Zambonelli. Engineering mobile agent applications via context-dependent coordination. *IEEE Transactions on Software Engineering*, 28 (11): 1039–1055, 2002.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

[11] C. K. Hess and R. H. Campbell. An application of a context-aware file system. *Personal and Ubiquitous Computing*, 7 (6): 339–352, 2003.

[12] A. Spriestersbach, H. Vogler, F. Lehmann, and T. Ziegert. Integrating context information into enterprise applications for the mobile workforce: a case study. In *Proceedings of the 1st International Workshop on Mobile Commerce*, pages 55–59. ACM Press, New York, 2001.

[13] S. S. Yau, S. K. S. Gupta. F. Karim, S. Ahamed, Y. Wang, and B. Wang. A smart classroom for enhancing collaborative learning using pervasive computing technology. In *Proceedings of the 6th WFEO World Congress on Engineering Education and the 2nd ASEE International Colloquium on Engineering Education* (*ASEE 2003*), pages 21–30. ACM Press, New York, 2003.

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

[15] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1992.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] S. S. Yau and F. Karim. Context-sensitive distributed software development for ubiquitous computing environments. In *Proceedings of the 25th Annual International Computer Software and Applications Conference* (*COMPSAC 2001*), pages 263–268. IEEE Computer Society Press, Los Alamitos, California, 2001.

[23] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36 (7): 75–84, 1993.

[24] S. S. Yau, Y. Wang, D. Huang, and H. P. In. Situation-aware contract specification language for middleware for ubiquitous computing. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems* (*FTDCS 2003*), pages 93–99. IEEE Computer Society Press, Los Alamitos, California, 2003.

[25] S. S. Yau and F. Karim. An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *Journal of Real-Time Systems*, 26 (1): 29–61, 2004.

[26] 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.

[27] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2 (3): 208–215, 1976.

[28] B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3 (3): 254–270, 1994.

[29] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4): 465–470, 1982.

[30] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), pages 327–333. IEEE Computer Society Press, Los Alamitos, California, 2002.

[31] 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.

[32] R. H. Carver and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24 (6): 471–490, 1998.

[33] B. Karacali and K.-C. Tai. Automated test sequence generation using sequencing constraints for concurrent programs. In *Proceedings of the 4th International Symposium on Software Engineering for Parallel and Distributed Systems* (*PDSE '99*), pages 97–108. IEEE Computer Society Press, Los Alamitos, California, 1999.

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**