Postprint of article in The Symposium on Engineering Test Harness (TSETH '13),
Proceedings of the 13th International Conference on Quality Software (*QSIC '13*),
IEEE Computer Society, Los Alamitos, CA (2013)

Oracles are Hardly Attain'd, And Hardly Understood: * Confessions of Software Testing Researchers

W.K. Chan
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
Email: wkchan@cityu.edu.hk

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

***Abstract*—In software testing, a test oracle refers to the mechanism for determining whether the results of the software under test agree with the expected outcomes. To achieve this, we need a means to determine the expected outcomes, a means to gauge the actual results, and a means to decide whether the actual results agree with the expected outcomes. In real-life situations, however, a test oracle may not exist owing to a missing link in any of these aspects. In this paper, we summarize our research for the last 15 years on selected issues related to each of these aspects. We present the use of metamorphic testing, pattern classification, and formal object equivalence and nonequivalence to alleviate the problems.**

Keywords*—*Test oracle, test harness, metamorphic testing, pattern classifier, object equivalence and nonequivalence

I. Introduction

The word "oracle" is often used to mean a prophecy or prediction revealed by a priest. In software testing, a *test oracle* [29] refers to the mechanism for determining whether the results of the software under test agree with the expected outcomes. It is an essential component in a test harness because the latter not only needs to execute test cases but also to report whether the test results are failures. In theory, test oracles can be determined by the software specification. In practice, however, the mechanism may not exist or may be too difficult or too costly. In such situations, we say that there is a *test oracle problem*. Programs with a test oracle problem are sometimes said to be *non-testable* [29].

There are, in fact, three assumptions behind the concept of test oracles:

(a) There is a mechanism to determine the expected outcomes.

(b) There is a mechanism to gauge the actual results.

(c) Given (a) and (b), there is a further mechanism to decide whether the actual results agree with the expected outcomes.

We have been conducting research on the test oracle problem and have addressed issues related to each of the three mechanisms above. In this paper, we summarize our research results for the last 15 years on selected topics.

II. Software Testing in the Absence of Expected Outcomes

A. Metamorphic Testing

A classic technique to test a function in the absence of a test oracle is to check whether the function preserves some expected mathematical identities [14], such as $\sin(\pi - x) = \sin x$. In [29], the idea was generalized to the testing of other identity relations determined from numerical or scientific theory. When some practitioners such as accountants prepare financial statements in spreadsheets, they may also match summations in columns and rows to ensure that values are not misplaced.

More recently, *metamorphic testing* was proposed in [11]. Intuitively, regardless of whether a test case by itself exposes any anomaly, it still contains useful information. To harvest such useful but hidden information for identifying program failures, given the original test case(s) and their expected results (if any), follow-up test case(s) can be constructed with reference to some necessary conditions relevant to the problem or some necessary properties of the algorithm in question. These conditions or properties are known as *metamorphic relations*.

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410, 123512, 716612, and 717811) and a linkage grant of the Australian Research Council (project no. LP100200208).

* William Shakespeare, *Henry VI*, Part II, Act I, Scene 4 (1590–91).

Let f be a target function to be implemented. Intuitively, a metamorphic relation is a necessary condition over a series of inputs $x_1, x_2, \ldots, x_n$ and their expected results $f(x_1), f(x_2), \ldots, f(x_n)$ for multiple evaluations of f. Metamorphic testing involves the verification of metamorphic relations. We adopt the definitions of metamorphic relation and metamorphic testing from [4] as follows:

Definition 1: (Metamorphic Relation). Let $x_1, x_2, \ldots, x_k$, where $k \geq 1$, be a series of inputs to a target function f and let $\langle f(x_1), f(x_2), \ldots, f(x_k)\rangle$ be the corresponding series of expected results. Suppose $\langle f(x_{i1}), f(x_{i2}), \ldots, f(x_{im})\rangle$ is a sub-series, possibly an empty sub-series, of $\langle f(x_1), f(x_2), \ldots, f(x_k)\rangle$. Let $\langle x_{k+1}, x_{k+2}, \ldots, x_n\rangle$, where $n \geq k+1$, be another series of inputs to f and let $\langle f(x_{k+1}), f(x_{k+2}), \ldots, f(x_n)\rangle$ be the corresponding series of expected results. Suppose, further, that there exist relations $r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$ and $r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$ such that r' must be true whenever r is satisfied. We say that

$$\begin{aligned}\boldsymbol{MR} = \{&(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))| \\ &r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), \\ &\quad x_{k+1}, x_{k+2}, \ldots, x_n) \\ &\rightarrow r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))\}\end{aligned}$$

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

$$\begin{aligned}\boldsymbol{MR:}\ \ &\text{If } r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), \\ &\quad x_{k+1}, x_{k+2}, \ldots, x_n), \\ &\text{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)).\end{aligned}$$

Definition 2: (Metamorphic Testing). Let P be an implementation of the target function f. *Metamorphic testing* of the metamorphic relation ***MR*** involves the following steps: (1) Given a series of *original test cases* $\langle x_1, x_2, \ldots, x_k\rangle$, generate a series of *follow-up test cases* $\langle x_{k+1}, x_{k+2}, \ldots, x_n\rangle$ according to the relation $r(x_1, x_2, \ldots, x_k, P(x_{i1}), P(x_{i2}), \ldots, P(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$. (2) Check whether relation $r'(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots x_n, P(x_1), P(x_2), \ldots, P(x_k), P(x_{k+1}), \ldots, P(x_n))$ is satisfied by the implementation P. If r' is evaluated to be false, then the metamorphic testing of *MR* reveals a failure.

It is obvious from Definition 1 that a metamorphic relation is not limited to an identity relation. In the next section, we will give an example of the application of metamorphic testing to the convergence property in the solution of partial differential equations.

We should add that the metamorphic approach is not only applicable to numerical programs but also to non-numerical software. For example, we have been awarded a Virtual Earth Award by Microsoft Research, Richmond, WA, USA for our application of metamorphic testing to Web search engines with map-related, search-related, and route-finding-related geographic components [33]. MT was also successfully applied to services computing [6]. We also find that metamorphic relations can be usefully applied to proving and debugging [13].

B. Application of Metamorphic Testing to Numerical Software

Although many attempts have been made to solve partial differential equations analytically, they are successful only in a limited number of applications. In practice, such equations are solved by numerical computation [8]. We encounter the oracle problem when testing programs that implement numerical methods because, in the absence of analytical solutions, the expected outcomes are usually not known.

In this section, we will illustrate how metamorphic testing can be applied to a program for solving partial differential equations despite the absence of expected outcomes. The original program was adapted from [15]. The testing process is adapted from [12].

We would like to determine the temperature distribution at every point on a heated square plate, which is a practical problem in thermodynamics. The plate is by itself a closed system, that is, there is no heat exchange with the environment. The heat distribution on the plate has reached a stable state. The temperature along each edge of the plate is homogeneous and is given.

The temperature T at any point P can be modeled by a Laplace equation

$$\frac{\partial^2 T}{\partial x^2}(P) + \frac{\partial^2 T}{\partial y^2}(P) = 0,$$

with fixed (or Dirichlet) boundary conditions [26]. The second derivatives can be approximated using the central difference technique:

$$\begin{aligned}\frac{\partial^2 T}{\partial x^2}(P) &\approx \frac{T(P_L) - 2T(P) + T(P_R)}{h^2} \\ \frac{\partial^2 T}{\partial y^2}(P) &\approx \frac{T(P_A) - 2T(P) + T(P_B)}{h^2}\end{aligned}$$

where P_A and P_B are points at a small distance h above and below P, and P_L and P_R are points at the same distance h to the left and right of P.

We will seed a fault into the program to demonstrate the challenge in the test oracle. Suppose we replace the statement

```
if fabs(uMat[i][j] − vMat[j][i]) > larg)
    larg = fabs(uMat[i][j] − vMat[j][i]);
```

by

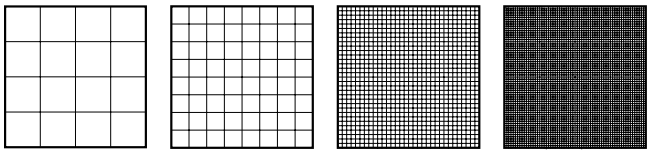
```
if fabs(uMat[i][j] − uMat[j][i]) > larg)
    larg = fabs(uMat[i][j] − vMat[j][i]);
```

In other words, we replace the variable vMat in the predicate by *uMat*.

Following common practices, we have tested the program using special test cases with known results. They include the following:

(a) The temperatures on all the four edges are the same. In this case, the temperature at all points on the plate will be identical.

(b) The boundary conditions are symmetrical with respect to the horizontal axis or the vertical axis or both. In this case, the resulting temperatures will also be symmetrical.



(a) G_1: 4 × 4 Mesh Grid (b) G_2: 8 × 8 Mesh Grid (c) G_3: 16 × 16 Mesh Grid (d) G_4: 32 × 32 Mesh Grid

Fig. 1. Improving precision by refining the mesh grids.

Unfortunately, the faulty program shows no failure for such special cases. We would like to apply metamorphic testing to alleviate the problem.

The convergence of numerical solutions for partial differential equations has been studied extensively in numerical analysis. For instance, according to the Lax-Richtmyer equivalence theorem [26], a consistent finite difference method for linear partial difference equations such as Laplace equations will converge if and only if it is stable. In any case, such theoretical analysis of the application domain is beyond the scope of the software tester. Thus, in the thermodynamics example, we will concentrate on the detection of failures in the implementation in relation to the specification by the chemical engineer that the stability condition of the Laplace equation has been satisfied and hence the numerical solution is expected to converge.

Given this preamble, we will formulate convergence as a metamorphic relation. We divide the plate uniformly using mesh grids as follows:

$$\begin{array}{ll} G_1: & 4\times 4 \text{ mesh grid,} \\ G_2: & 8\times 8 \text{ mesh grid,} \\ G_3: & 16\times 16 \text{ mesh grid,} \\ G_4: & 32\times 32 \text{ mesh grid, and} \\ G_5: & 64\times 64 \text{ mesh grid.} \end{array} \tag{1}$$

The step size h_i for the grid G_i is given by $h_1 = 2h_2 = 4h_3 = 8h_4 = 16h_5$. The first four mesh grids are shown in Fig. 1. We write $G_1 \prec G_2 \prec \ldots \prec G_5$.

Let $T(P)$ be the expected solution of the partial differential equation at P. Let $T_{G_i}(P)$, $T_{G_j}(P)$, and $T_{G_k}(P)$ be the temperatures at point P determined through the mesh grids G_i, G_j, and G_k, respectively. Because of the convergence property as determined by the chemical engineer, as the densities of the mesh grids increase, the expected errors will decrease. Hence, the temperatures at point P computed by the respective grids will satisfy the following necessary condition:

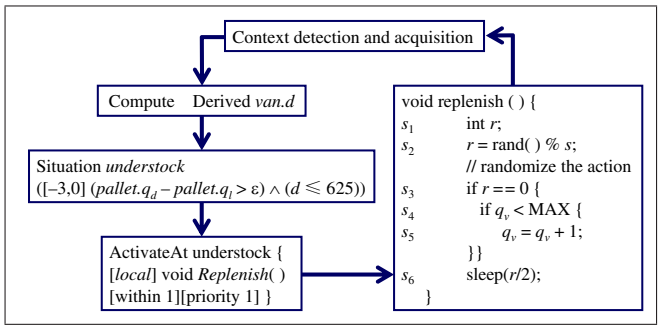
$$\begin{array}{l} \text{If } G_i \prec G_j \prec G_k, \\ \text{then } \left|T_{G_k}(P) - T(P)\right| \le \left|T_{G_j}(P) - T(P)\right| \le \left|T_{G_i}(P) - T(P)\right|. \end{array}$$

However, the expected value of $T(P)$ is not known in the absence of a test oracle. To solve the problem, we have proven in [12] that we can eliminate $T(P)$ from the equation, giving the following metamorphic relation:

$$\begin{array}{ll} \boldsymbol{MR_{PDE}}\textbf{:} & \text{If } G_i \prec G_j \prec G_k, \\ & \text{then } T_{G_i}(P) \le \min\{T_{G_j}(P),\ T_{G_k}(P)\} \text{ or} \\ & \quad T_{G_i}(P) \ge \max\{T_{G_j}(P),\ T_{G_k}(P)\}. \end{array}$$

TABLE I. OUTPUTS FROM PARTIAL DIFFERENTIAL EQUATION PROGRAM.

Point	T_{G_1}	T_{G_2}	T_{G_3}	T_{G_4}	T_{G_5}
P_1	107.857	106.716	106.344	106.272	105.603
P_2	105.893	104.789	104.376	104.298	103.357
P_3	75.714	74.126	73.624	73.518	72.841
P_4	175.536	174.056	173.472	173.343	172.389
P_5	190.000	190.000	**189.925**	**189.953**	**188.657**



(a) Interaction between middleware (top) and software application (right) via SA-IDL specification (left)

Situation *understock*
([–3,0] (*pallet.q_d* – *pallet.q_l* > ε) ∧ (**d ≤ 265**))

(b) Faulty SA-IDL specification

Fig. 2. Example interaction in context-sensitive middleware-based application.

A failure is revealed if a set of original and follow-up test cases do not satisfy this relation.

We have run the program over five sample points $P_1, P_2, \ldots, P_5$ using the mesh grids in (1). The resulting temperatures are shown in Table I. Consider, in particular, the point P_5. We find that, even though $G_3 \prec G_4 \prec G_5$, we have $T_{G_5}(P_5) < T_{G_3}(P_5) < T_{G_4}(P_5)$. This contradicts the metamorphic relation $\boldsymbol{MR_{PDE}}$. Hence, a failure is revealed.

III. SOFTWARE TESTING WHEN ACTUAL RESULTS ARE NOT COMPLETELY OBSERVABLE

A. Testing of Ubiquitous Computing Applications

To efficiently and effectively organize factories, warehouses, and fleets in supply chain management, logistics software applications that can configure smartly without human intervention play a crucial role. An essential enabling technology is ubiquitous middleware-based sensor network systems.

Ubiquitous computing, which means computing everywhere and at any time, is in high demand today in emerging mobile applications. Context-sensitive concurrent middleware-based software is an emerging hot topic in ubiquitous computing [2], [3], [20], [25], [30], [31], [32]. A *context* is an instantaneous attribute of the environment relevant to the application, such as stock levels and traffic conditions. *Context-sensitive* software dynamically adapts its operations according to the changing environment in an attempt "to move from current styles of rigid and context-free human-machine interaction" [23]. In *middleware-based* software, the actual process of

accessing and updating the contexts lies with the middleware. Whenever the contexts inscribed in the context-aware interface are satisfied, the middleware invokes the relevant local and remote operations of the software applications atop.

This approach not only frees applications from the tedious need to interpret raw sensor data into contexts, but also provides the middleware with an opportunity to optimize available resources for each application. Nonetheless, the lack of sensor data controls at the application level results in an incomplete knowledge of environmental information and an unpredictable behavior of the middleware in optimizing its resources. This limits the ability of an application to react to external stimuli and to coordinate its devices to achieve desirable effects. Such intrinsic limitations of typical context-sensitive concurrent middleware-based systems make it difficult for the software to behave correctly. At the same time, since both correct and abnormal behaviors may be blurred by the incomplete knowledge of environmental information, it also makes the faults in the software more difficult to be revealed. Thus, various researchers (such as [1]) have emphasized the difficulties in assuring the quality of ubiquitous computing applications.

According to Rosenblum et al. [24], we published the first work [28] on testing techniques for ubiquitous systems. We noted that the contexts being used or updated by such systems are observable and possess specific properties in the context spaces surrounding the applications. Thus, the usages of and changes in contexts provide invaluable sources for test adequacy and test oracle information to assure the correctness of ubiquitous software applications. We will concentrate on the test oracle issue on ubiquitous computing applications in the remaining parts of this section. [1]

B. Unit Testing in Ubiquitous Computing

Context detections and function activations are conducted by the middleware. 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-paths" coverage in white-box testing), or to use the contexts registered in the middleware as activation conditions. Furthermore, it is an extremely difficult task to work out a precise oracle for testing the application.

Consider an example from [28] regarding a smart delivery system in a supermarket chain. Every pallet (or tray) is configured to store a particular kind of product at a desired stock level. When the stock level is high, no replenishment is required. When the level is low, the smart pallet will automatically request delivery vans to replenish the products.

Every delivery van handles a type of product. The effective delivery distance by any van to any pallet is at most 1 km. A smart pallet will detect a suitable delivery van when it moves within the effective distance, and will request for replenishment if the desired stock level is not met. The replenishment signal may also be sensed by any other delivery vans nearby. The latter will not take replenishment actions if the closest van acknowledges the replenishment request. Unfortunately, this van may not be able to deliver the requested quantity to a particular pallet if other neighboring pallets also ask for replenishment at the same time.

In particular, the situation *understock* represents that a smart pallet is within the effective delivery zone at time t and that the current ledger amount q_l of the pallet is short of the desired quantity of q_d for more than a tolerance level of ε in the past 3 seconds. When this happens, the application will replenish the product items on the pallet. This is achieved by calling the local function *replenish*(). The situation *overstock* is specified in the same manner.

The interface between an application and the middleware is defined in the reconfigurable context-sensitive middleware framework [31] by means of a formal SA-IDL specification, as illustrated in the supermarket example in Fig. 2(a). The specification is automatically translated into code by a SA-IDL compiler. Suppose there is a fault in the SA-IDL specification of the detection device in delivery vans. In the situation expression *understock*, the predicate $d \leq 625$ as shown in Fig. 2(a) has been mistakenly coded as $d \leq 265$ as in Fig. 2(b). As a result, the fault in the specification is also carried forward to the program code. There are several challenges in testing the application in the presence of interactions with the middleware:

(a) Since the middleware is continuously interacting with the application according to changing contexts in the environment, it is difficult to determine an appropriate time for checking the "final" result of a test case.

(b) Since the activity *replenish*( ) is non-deterministically invoked, it will take an indefinite number of invocations before the situation *understock* is no longer applicable. Thus, it is very difficult to tell the difference between a successful and a failed execution.

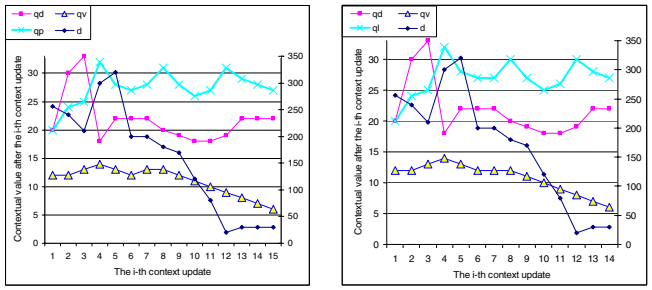
(c) At the unit testing level, because of the simplicity of *replenish* module, we can easily construct two simple test cases to fulfil the *all-branches* criterion for control-flow coverage, the *all-du-paths* criterion for data-flow coverage and the *all-predicate-use* criterion for predicate-based testing. However, no failure can be revealed.

On the other hand, our studies [28] showed that metamorphic relations are an effective means to overcome these challenges. Owing to the contention among smart pallets and delivery vans within the neighborhood, the actual stock in a pallet may differ a great deal from its desired level. It would, therefore, be too restrictive to use the desired level as a test oracle to check against the final stock level attained. We proposed to use isotropic properties of environmental contexts as metamorphic relations for unit testing of context-sensitive software. Relations among multiple input/outputs of the applications and environmental variables were used as the correctness criteria.

For instance, the application should provide consistent stock levels to different pallets in similar situations throughout the supermarket chain. The following, therefore, is an intuitive and yet effective metamorphic relation for identifying the failure due to the faulty program above:

> $\boldsymbol{MR_{SuperMkt1}}$: Let TC be an original test case and TC' be a follow-up test case. If the distance between the pallet and the van for TC is comparable to that for TC', the ledger quantities q_l for both test cases should be comparable.

[1] Readers may refer to [16], [17], [18], [19], [21], [22] for our research on the test adequacy issue in general.



(a) Expected outcomes (b) Observed results

Fig. 3. Spot the difference: Challenge of obscured failures in integration testing of ubiquitous computing applications.

C. Integration Testing in Ubiquitous Computing

During integration testing of conventional software, even though various components may interact with one another, the behavior and outcomes of the application are determined within the implemented system. On the other hand, context detections and function activations of a context-sensitive middleware-based application are the responsibilities of the middleware.

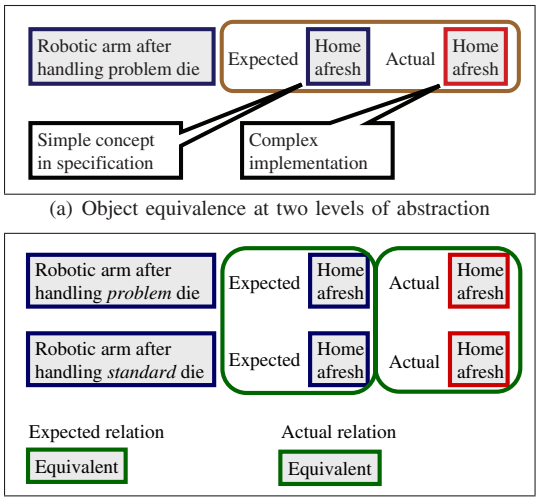
Thus, in integration testing, the middleware may repeatedly invoke various software components according to the interface contexts, until the triggering conditions inscribed in the middleware are no longer satisfied. In this way, the middleware may remain active and continue to trigger further actions according to the renewed situations, so that the termination of test cases may not be determined. Furthermore, a failure caused by an invoked function may immediately be superseded by a subsequent correct action, so that the former cannot be observed by testers. Interested readers may compare Figs. 3(a) and 3(b) to appreciate the challenge due to obscured failures.

We propose to exploit a special kind of situation, which we call checkpoints, during which the middleware is temporarily *not* activating any function under test. Testers should generate test cases that start at one checkpoint and end at another. This would free the termination of the test cases from being affected by the middleware. On the other hand, various functions are still being invoked during the period between two checkpoints, so that the integration testing can be reasonably conducted. Testers may propose metamorphic relations that associate different operation sequences of a test case between two checkpoints, with a view to detecting whether such relations are infringed by the application under test.

For integration testing of the supermarket example in the last section, the following metamorphic relation may be used:

> ***MR*$_{\textbf{SuperMkt2}}$**: Let TC be an original test case and TC' 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 TC', then the number of invocations of the *replenish*( ) function for TC' is expected to be more than that of TC. If we apply *replenish*( ) to the initial checkpoint before executing TC', then the number of invocations of the *withdraw*( ) function for TC' is expected to more be than that of TC.

Readers may refer to [4] for details.



(a) Object equivalence at two levels of abstraction

(b) Object equivalence at same level of abstraction

Fig. 4. Challenge of object equivalence at different levels of abstraction.

IV. Software Testing without a Straightforward Notion of Equivalence

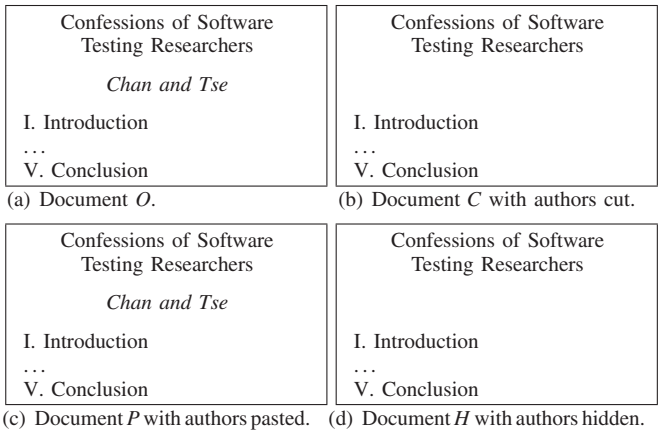
A. Testing of Object-Oriented Software

Object-oriented programming is considered to be the most popular programming paradigm because of the close resemblance of the data and program structures with the real world being modeled. Object-oriented software typically consists of classes of objects with their own properties, behaviors, and interactions with one another. The fundamental notions such as abstraction, encapsulation, inheritance, and polymorphism enable object-oriented software to be more flexible, maintainable, and reusable. On the other hand, object-oriented programming poses challenges to testing, since the classes of objects may interact with one another with unforeseen combinations, possibly with hidden attributes and methods. They are much more complex to simulate and test than the hierarchy of modules in conventional programs.

In particular, in program testing, we check whether the actual result produced by the software is the same as the expected outcome. What exactly is meant by "the same"? This is no easy question to answer when testing object-oriented software.

Consider, for instance, a robotic arm R_1 in an automatic assembly system for semiconductor chips. Suppose it encounters a problem when handling a chip. It is expected to discard the die and return to the home position. If the software that controls the arm is correct, the actual arm should do exactly the same thing. This is exactly what we need to test: Is the actual behavior of the robotic arm object "the same" as that of that of the expected behavior?

We note, however, that a real-life specification may only describe object behavior at a high level and does not include all the implementation details. In other words, the expected object may involve abstract concepts at the specification level while the actual object involves complex implementation issues, as illustrated in Fig. 4(a). Hence, it is not feasible to compare the equivalence of these two objects.



(a) Document O. (b) Document C with authors cut.

(c) Document P with authors pasted. (d) Document H with authors hidden.

Fig. 5. Examples of equivalence of objects.

We would like to mimic metamorphic testing in the previous sections. Consider a second robotic arm R_2 that has not encountered a problem when handling a chip. After completing the processing of the chip, it should also return to the home position. In other words, both robotic arms R_1 and R_2 should exhibit the same behavior as soon as they have returned to their home positions. They should forget about whether they were successful in processing the last chip. This is summarized diagrammatically in Fig. 4(b). We say that two objects are *Observationally Equivalent* (*OE*) if they exhibit exactly the same behavior, that is, they show the same visible results when subjected to the same sequence of operations. Otherwise, we say that they are *Observationally Nonequivalent* (OE').

An implementation is consistent with the specification if and only if both of the following criteria are satisfied:

> ***Equivalence Criterion.*** For any two sequences of operations that are OE according to the specification, the resulting objects in the implementation must be OE.
>
> ***Non-Equivalence Criterion.*** For any two sequences of operations that are OE$'$ according to the specification, the resulting objects in the implementation must be OE$'$.

We say there is a failure in the implementation if it is not consistent with the specification.

OE appears to be the most intuitive means of verifying whether two objects are equivalent. It is not practical in real life, however, because OE cannot be checked easily. Infinitely many operation sequences may be applied to an object, so that it will be impossible for testers to confirm the OE of even one pair of objects. We would like, therefore, to explore other forms of equivalence.

Consider an example of a Microsoft Word Document O as shown in Fig. 5(a). The authors thought that double-blind reviews were necessary. Hence, they cut their author names, producing Document C as shown in Fig. 5(b). They were then advised that single-blind would be used, so that they pasted back their names, producing Document P as shown in Fig. 5(c). The question is, are Documents O and P equivalent? We give a dump of both documents and found that they are not identical objects. However, we find they exhibit exactly the same behavior, producing exactly the same visible results under the same sequence of operations.

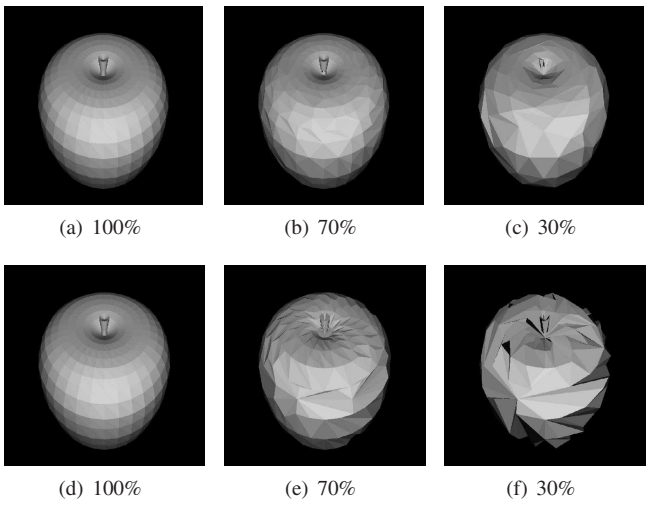
Alternatively, instead of using "cut", the authors may use the "hidden" option in the font menu to hide their names. Document H will result, as shown in Fig. 5(d). Are Documents C and H equivalent? We find that they appear the same on the screen and when printed. However, they exhibit different behaviors. For example, we can paste the name back to Document C, but there is nothing to be pasted for Document H. On the other hand, we may press the "show" button to see the hidden text in H but not in C.

In summary, we may have three levels of equivalence: The concept of *identical objects* is very easy to check, but is generally too strong to be useful in object-oriented program testing. It does not take into account the notion of encapsulation in object-oriented software, where some of the attributes and behavior may be hidden and should not be used as a basis for verifying the equivalence. *Attribute Equivalence* (*AE*), which compares only the visible attributes of two objects like C and H above, is generally very easy to check but is too weak to be useful in testing. *Observational Equivalence* (*OE*) is what we need in object-oriented software testing, but it is generally too difficult to check.

We have three corresponding levels of equivalence among sequences of operations in software specifications: Like identical objects, *Normal Equivalence* (*NE*) of two sequences of operations is very easy to check but is generally too strong to be useful in specification-based testing of object-oriented programs because it does not take into account the concept of encapsulation. *Attribute Equivalence* (*AE*) is generally very easy to check but is too weak to be useful in testing. *Observational Equivalence* (*OE*) is what we need in specification-based testing, but it is generally too difficult to check.

We attempted to find solutions to this serious practical problem. We discovered and proved the following two important theorems for specifications and implementations that satisfy specific fundamental conditions, namely, that the algebraic specification of the classes is canonical with proper imports and the implementation is complete. In this way, the difficult tasks of checking OE and OE$'$ in the general situation can be alleviated.

> *Theorem 1: (Equivalence Criteria)* (adopted from [10]). Given a canonical specification of a class with proper imports, suppose its implementation is complete. The following statements imply one another:
>
> (a) For any two sequences of operations that are OE according to the specification, the corresponding objects in the implementation must be OE.
>
> (b) For any two sequences of operations that are NE according to the specification, the corresponding objects in the implementation must be OE.
>
> (c) For any two sequences of operations that are NE according to the specification, the corresponding objects in the implementation must be AE.
>
> (d) For any two sequences of operations that are AE according to the specification, the corresponding objects in the implementation must be AE.
>
> (e) For any two sequences of operations that are OE according to the specification, the corresponding objects in the implementation must be AE.



(a) 100% (b) 70% (c) 30%

(d) 100% (e) 70% (f) 30%

Fig. 6. Mesh simplification of polygonal models of a properly rendered apple and a badly rendered apple (from [5].)

> *Theorem 2: (Non-Equivalence Criteria)* (adopted from [10]). Given a canonical specification of a class with proper imports, suppose its implementation is complete. The following statements imply one another:
>
> (a) For any two sequences of operations that are OE′ according to the specification, the resulting objects in the implementation must be OE′.
>
> (b) For any two sequences of operations that are AE′ according to the specification, the resulting objects in the implementation must be AE′.
>
> (c) For any two sequences of operations that are AE′ according to the specification, the resulting objects in the implementation must be OE′.

In this way, we can make use of the straightforward AE and AE′ to replace the impossible task of checking OE and OE′ in object-oriented software testing. Based on the theoretical results, we have developed approaches to generate equivalent and nonequivalent objects as test cases. The results turn out to be practically viable [9], [10], [27].

B. Testing of Graphics Applications

During the testing of multimedia and graphics rendering applications, it is also challenging to compare actual graphics outputs with expected results. Since the expected outcome may not be precisely defined, automatic pixel-by-pixel comparison is out of the question. Testers usually resolve to manual checking, which is labour-intensive and error-prone. See Fig. 6, for example. It would be useful if an automatic pseudo-oracle would be found.

We propose the use of *reference model*, which is an existing program that serves the same purpose and/or specification as the software under test, but not necessarily implemented using the same algorithm. We then train a pattern classifier to recognize the black-box features of samples from the reference model and its fault-based variations.

We also pipe the test cases marked as "passed" by the pattern classifier to a second metamorphic testing component. Simple graphics properties are used as metamorphic relations to look for missed failures. For example, if the input points are flipped or scaled, the output image should be flipped or scaled accordingly.

Our experiments [5], [7] show that this approach significantly enhances the failure detection effectiveness of pattern classification.

V. Conclusion

A test oracle is an essential component in a test harness because the latter not only needs to execute test cases but also to report whether the test results are failures. This paper summarizes our work published in journals and conferences for the last 15 years on selected issues related to software testing in the absence of a precise test oracle. We have addressed three aspects of the problem:

(a) ***Testing without a Mechanism to Determine the Expected Outcomes.***

We introduced the concept of metamorphic testing and illustrated our approach through a numerical application for the solution of partial differential equations. The technique may also be applicable to non-numerical situations such as Web search engines with geographic components.

(b) ***Testing without a Mechanism to Gauge the Actual Results.***

Ubiquitous context-sensitive middleware-based software applications are difficult to test because the changing oracles are too short-lived to be verified manually or automatically using conventional techniques. We have adopted the metamorphic testing approach to handle these systems.

(c) ***Testing without a Mechanism to Decide Whether the Actual Results Agree with the Expected Outcomes.***

In object-oriented testing, testers often need to verify whether the actual object produced by the software is the same as the expected test oracle. Since an infinite combination of operations may affect the behavior of an object, checking the outputs of the object at each state for all such operation sequences is too complex. We have developed practical criteria for identifying failures in the equivalence and nonequivalence of objects. We have also tackled a similar problem in graphics rendering software using pattern classifiers, and integrated them with metamorphic testing.

Acknowledgements

We are grateful to our project partners F.T. Chan, H.Y. Chen, T.Y. Chen, S.C. Cheung, J. Feng, M. Hagenbuchner, J.C.F. Ho, B. Jiang, F.-C. Kuo, Z. Lai, F.C.M. Lau, K.R.P.H. Leung, P.C.K. Liu, H. Lu, C.K.F. Luk, L. Mei, B. Xu, C. Xu, S.S. Yau, C. Ye, K. Zhai, S. Zhang, Z. Zhang, and Z.Q. Zhou for their excellent contributions to our research.

References

[1] G. Banavar and A. Bernstein, "Software infrastructure and design challenges for ubiquitous computing applications," *Communications of the ACM*, vol. 45, no. 12, pp. 92–96, 2002.

[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*, vol. 29, no. 12, pp. 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*, vol. 29, no. 12, pp. 1072–1085, 2003.

[4] W.K. Chan, T.Y. Chen, H. Lu, T.H. Tse, and S.S. Yau, "Integration testing of context-sensitive middleware-based applications: a metamorphic approach," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 5, pp. 677–703, 2006.

[5] W.K. Chan, S.C. Cheung, J.C.F. Ho, and T.H. Tse, "PAT: a pattern classification approach to automatic reference oracles for the testing of mesh simplification programs," *Journal of Systems and Software*, vol. 82, no. 3, pp. 422–434, 2009.

[6] W.K. Chan, S.C. Cheung, and K.R.P.H. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research*, vol. 4, no. 2, pp. 60–80, 2007.

[7] W.K. Chan, J.C.F. Ho, and T.H. Tse, "Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 89–120, 2010.

[8] S.C. Chapra and R.P. Canale, *Numerical Methods for Engineers*, McGraw-Hill, 2006.

[9] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen, "In black and white: an integrated approach to class-level testing of object-oriented programs," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 250–295, 1998.

[10] H.Y. Chen, T.H. Tse, and T.Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 56–109, 2001.

[11] T.Y. Chen, S.C. Cheung, and S.M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Technical Report HKUST-CS98–01, 1998.

[12] T.Y. Chen, J. Feng, and T.H. Tse, "Metamorphic testing of programs on partial differential equations: a case study," *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 02*), IEEE Computer Society, 2002, pp. 327–333.

[13] T.Y. Chen, T.H. Tse, and Z.Q. Zhou, "Semi-proving: an integrated method for program proving, testing, and debugging," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 109–125, 2011.

[14] W.J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

[15] C.F. Gerald and P.O. Wheatley, *Applied Numerical Analysis*, Addison-Wesley, 1999.

[16] B. Jiang, K. Zhai, W.K. Chan, T.H. Tse, and Z. Zhang, "On the adoption of MC/DC and control-flow adequacy for a tight integration of program testing and statistical fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 897–917, 2013.

[17] Z. Lai, S.C. Cheung, and W.K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesC applications," *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 08/FSE-16*), ACM, 2008, pp. 94–104.

[18] H. Lu, W.K. Chan, and T.H. Tse, "Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation," *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 06/FSE-14*), ACM, 2006, pp. 242–252.

[19] H. Lu, W.K. Chan, and T.H. Tse, "Testing pervasive software in the presence of context inconsistency resolution services," *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 08*), ACM, 2008, pp. 61–70.

[20] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, "Xmiddle: a data-sharing middleware for mobile computing," *Wireless Personal Communications*, vol. 21, no. 1, pp. 77–103, 2002.

[21] L. Mei, W.K. Chan, and T.H. Tse, "Data flow testing of service-oriented workflow applications," *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 08*), ACM, 2008, pp. 371–380.

[22] L. Mei, W.K. Chan, and T.H. Tse, "Data flow testing of service choreography," *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 09/FSE-17*), ACM, 2009, pp. 151–160.

[23] 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*, vol. 47, no. 1, pp. 51–56, 2004.

[24] D.S. Rosenblum, C. Mascolo, M.Z. Kwiatkowska, D. Ghica, M. Ryan, N. Dulay, and E. Lupu, "UbiVal: fundamental approaches to validation of ubiquitous computing applications and infrastructures," University of Birmingham, University College London and Imperial College London, UK, Project Proposal, EPSRC Project GR/D076625/01, 2006–2010.

[25] M. Sama, D.S. Rosenblum, Z. Wang, and S.G. Elbaum, "Model-based fault detection in context-aware adaptive applications," *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 08/FSE-16*), ACM, 2008, pp. 261–271.

[26] J.C. Strikwerda, *Finite Difference Schemes And Partial Differential Equations*, Society for Industrial and Applied Mathematics, 2004.

[27] T.H. Tse, F.C.M. Lau, W.K. Chan, P.C.K. Liu, and C.K.F. Luk, "Testing object-oriented industrial software without precise oracles or results," *Communications of the ACM*, vol. 50, no. 8, pp. 78–85, 2007.

[28] T.H. Tse, S.S. Yau, W.K. Chan, H. Lu, and T.Y. Chen, "Testing context-sensitive middleware-based software applications," *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 04*), vol. 1, IEEE Computer Society, 2004, pp. 458–465.

[29] E.J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[30] C. Xu, S.C. Cheung, W.K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 3, p. article no. 9, 2010.

[31] S.S. Yau, F. Karim, Y. Wang, B. Wang, and S.K.S. Gupta, "Reconfigurable context-sensitive middleware for pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 33–40, 2002.

[32] F. Zambonelli, N.R. Jennings, and M. Wooldridge, "Developing multi-agent systems: the Gaia methodology," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 3, pp. 317–370, 2003.

[33] Z.Q. Zhou, S. Zhang, M. Hagenbuchner, T.H. Tse, F.-C. Kuo, and T.Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.