A Similarity Metric for the Inputs of OO Programs and Its Application in Adaptive Random Testing

Jinfu Chen, *Member, IEEE*, Fei-Ching Kuo, *Member, IEEE*, Tsong Yueh Chen, *Member, IEEE*, Dave Towey, *Member, IEEE*, Chenfei Su, and Rubing Huang, *Member, IEEE*

***Abstract*—Random testing (RT) has been identified as one of the most popular testing techniques, due to its simplicity and ease of automation. Adaptive random testing (ART) has been proposed as an enhancement to RT, improving its fault-detection effectiveness by evenly spreading random test inputs across the input domain. To achieve the even spreading, ART makes use of distance measurements between consecutive inputs. However, due to the nature of object-oriented software (OOS), its distance measurement can be particularly challenging: Each input may involve multiple classes, and interaction of objects through method invocations. Two previous studies have reported on how to test OOS at a single-class level using ART. In this study, we propose a new similarity metric to enable multiclass level testing using ART. When generating test inputs (for multiple classes, a series of objects, and a sequence of method invocations), we use the similarity metric to calculate the distance between two series of objects, and between two sequences of method invocations. We integrate this metric with ART and apply it to a set of open-source OO programs, with the empirical results showing that our approach outperforms other RT and ART approaches in OOS testing.**

***Index Terms*—Adaptive random testing (ART), method invocation, object distance, object-oriented software (OOS) testing, test input distance.**

Acronyms and Abbreviations

OOS	Object-oriented software.
RT	Random testing.
SUT	Software under test.
ART	Adaptive random testing.
RefV	Reference variable.
NRefV	Nonreference variable.
OBJ	Objects set.
MINV	Method invocations.
LenD	Length difference.
MsD	Method set difference.
SD	Sequence difference.
DT	Data types.
DO-ART	Divergence-oriented ART.
RT-1	RT with one method invocation.
RT-n	RT with multiple method invocations.
CL	Confidence level.
sDev	Standard deviation.

Notations

T	Executed test inputs set.
C	Candidate set.
τ, λ, ν, ϕ	Coefficients.
$weight\_a$, α	Coefficients.
$\mathbf{i}$	Imaginary unit.
A	Set of attributes.
N	NRefV attributes.
R	RefV attributes.
M	Set of methods.
p, q	Any two objects.
t	A test input.
$t.OBJ$	A list of objects in t.
$t.MINV$	An ordered list of methods in t.
$p.A.N[name_i]$	Value of variable $name_i$ in $p.A.N$.
pG	*NRefV* attributes sets for $p.A.N$.
DT_i	The ith data type.
$pG[i][j]$	The jth nonreference variable of $pG[i]$.
dist(x, y)	The distance between x and y.
F_m	The number of test inputs required to detect the first failure.
E_m	Expected number of failures detected.
F_m - time	Time taken to detect the first failure.
low$B$	Lower bound.
upp$B$	Upper bound.

Manuscript received June 20, 2015; revised December 16, 2015 and June 17, 2016; accepted November 8, 2016. Date of publication December 19, 2016; date of current version May 31, 2017. This work was supported in part by the National Natural Science Foundation of China under Grant 61202110 and Grant 61502205, in part by the Postdoctoral Science Foundation of China under Grant 2015M571687 and Grant 2015M581739, and in part by the Australian Research Council under Grant DP 120104773. Associate Editor: W. E. Wong.

J. Chen, C. Su, and R. Huang are with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China (e-mail: jinfuchen@ujs.edu.cn; suzenfly@163.com; rbhuang@ujs.edu.cn).

F.-C. Kuo and T. Y. Chen are with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia (e-mail: dkuo@swin.edu.au; tychen@swin.edu.au).

D. Towey is with the School of Computer Science, University of Nottingham Ningbo China, Ningbo 315100, China (e-mail: dave.towey@nottingham.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2016.2628759

I. Introduction

IT is well recognized that object-oriented (OO) design increases software reusability, extensibility, and interoperability [1], and because of this, we have witnessed an increasing amount of software being developed using OO programming languages. Such software (hereafter abbreviated as OOS) consists of classes, each of which is a design framework to create

multiple objects with common behavior and attributes, which are realized as *methods* and *instance variables*, respectively. *Constructors* are special methods used to create an object of a class, and to initialize its attributes. Objects can be coordinated to perform certain tasks in the OOS, and hence each test input for OOS often involves multiple classes and object interactions through *method invocations*.

Software testing is an important activity in software quality assurance, but OOS testing can be particularly challenging due to the special, but powerful features of OO languages, such as encapsulation, inheritance, and polymorphism [2], [3].

Although simple in concept, random testing (RT) is an important quality assurance technique [4]–[6], which can facilitate reliability estimation of the software under test (SUT), and, due to its simplicity and ease of automation, is commonly used in industry [7]. Many research studies have examined OOS testing using RT [5], [8]–[10], which can be easily applied to test at many levels (including unit, integration, and system levels) [5], [9], [10]. However, it has been noted that RT's effectiveness and efficiency deteriorate as the OOS size and complexity increase. Some studies have attempted to improve RT's performance in OOS testing [5], [7], [11], [12], an example of which is the development of feedback-directed RT, which uses information from executed test inputs to guide new test input generation. When information shows that certain parameter settings cause an exception, this technique will avoid such settings in subsequent RT test inputs.

In order to improve the failure-detection effectiveness of RT, Chen *et al.* proposed an enhancement, adaptive random testing (ART) [4], [13], [14], which was inspired by empirical observations that many faulty programs have contiguous regions of failure-causing inputs [15]. To quickly find a failure in this situation, ART selects randomly generated follow-up test inputs further away from the previously executed, nonfailure-causing test inputs. ART has been found to consistently outperform RT for fault detection [13], [15]. One challenge faced when applying ART is how to measure the difference between inputs (named "*distance*" hereafter). ART has been applied to testing OOS, with Ciupa *et al.* proposing a distance concept for objects, and a metric to measure differences between two objects of the same class, or derived from the same ancestor [16], [17]. They developed a tool, ARTOO, in the Eiffel language using this approach, and showed ARTOO to be better than RT, both in terms of number of test inputs to detect the first failure, and the number of uncovered faults. However, this metric can only choose one object at a time as a test input to test one specific method of the class. Lin *et al.* [18] tried to improve ARTOO by invoking extra methods of the chosen object before running the specific method. Using the Java language, they implemented this approach (called divergence-oriented ART) in a new tool called ARTGen, which was found to outperform RT in terms of finding faults.

A challenge facing ARTOO and divergence-oriented ART is that the distance metric may not readily allow generation and even spreading of test inputs involving multiple objects and methods of multiple classes. Although it can only be applied to pairs of objects, comparison of objects of different classes, or those not sharing a common ancestor is not possible. Furthermore, comparisons do not include object behavior information or complex object data (such as *enumerated* types, *arrays*, *structs*, and *pointers*).

In this paper, we propose a more generic distance metric, the object and method invocation sequence similarity (OMISS) metric, which facilitates integration testing of OOS. Integration testing requires that each test input be composed from multiple classes, and that a set of objects be created and able to interact with each other through a series of method invocations. When measuring inputs such as this, two levels of measurement are considered for developing the OMISS distance metric: object level and test-input level. The object level measures any pair of objects, regardless of which classes they are created from, and whether or not they share the same ancestor. The test-input level measures distances between object sets and method invocation sequences in the two inputs. These two measurement levels make the OMISS metric capable of handling complex inputs involving multiple classes, objects, and method invocations, and thereby facilitate integration testing. We have integrated the OMISS distance metric with the fixed-sized-candidate-set ART (FSCS-ART) [19] method to develop a testing tool, called OMISS-ART, for testing C++ and C# programs.

The remainder of this paper is organized as follows. The background information for OOS testing and ART is given in Section II. Our distance metrics are detailed in Section III. The development of OMISS-ART and procedures to generate test inputs are discussed in Section IV. Settings and results of our empirical studies are reported in Section V. Related work is discussed in Section VI, and the conclusion and future work are presented in Section VII.

II. Background

A. OO Software Testing

Software testing is an essential software quality assurance activity, but OOS testing can be particularly challenging due to the intricacies introduced by the special features of OO languages, such as encapsulation, inheritance, and polymorphism [1].

Test input design and generation are important aspects of software testing, directly impacting on the testing cost and effectiveness, and are often guided by things such as program structure, source code, the software specification, input/output structure, and information about previously executed tests [4], [20]. Depending on the information used, test input design and generation techniques have been broadly classified as specification based, structure based, and RT based [1], [21]. While both specification-based and structure-based approaches require detailed information about the OOS (either the specification or program structure), RT-based approaches have no such requirements, and can generate a large number of test inputs at low cost [1], [5], [8]–[10]. Furthermore, RT is conceptually simple, can easily be scaled, is readily applicable to many kinds of software, and has been widely used in industry [7].

RT techniques for unit testing of OOS have been attracting a lot of attention recently [5], [7], [11], [12], [22], [23], an example of which is Pacheco *et al.*'s [5], [7], [12] feedback-directed RT technique, which improves on the fault-detection effectiveness of RT by using information of previously executed test

Procedure 1: FSCS-ART algorithm based on max-min selection criterion and a distance metric.

```
1: INPUT distance metric DistMetric
2: Construct T = {} to store past executed test inputs
3: Randomly generate a test input t
4: Add t to T
5: while before reaching the stopping condition do
6:    Let maxD = 0.0
7:    Randomly generate a set of k candidates: C = {c1, c2,
      ..., ci, ..., ck}
8:    for each candidate ci in C do
9:       Let minD = MAX_VALUE {MAX_VALUE is
         the biggest value that a type of data can hold.}
10:      for each test input tj in T do
11:         Calculate the distance dj between ci and tj
            according to DistMetric
12:         if dj < minD then
13:            minD = dj
14:         end if
15:      end for
16:      if minD > maxD then
17:         maxD = minD
18:         t = ci
19:      end if
20:   end for
21:   Add t to T
22: end while
```

inputs to guide new input generation. According to this technique, if certain parameter settings lead to an exception, then inputs with these settings will be filtered out during the random generation, thereby lowering detection of duplicate failures. Studies have shown that feedback-directed RT improves the cost-effectiveness of RT in terms of detecting failures [5].

B. Adaptive Random Testing

ART [4], [14], [15], [19] has been proposed as an enhancement to RT. Inspired by many empirical observations that faulty programs usually have contiguous failure regions, ART aims to evenly spread the random test inputs across the input domain in order to enhance the failure-detection effectiveness.

ART has drawn a lot of attention, both from academia and from industry, and a number of different algorithms have been developed [13], [18], [19], [24]–[30], with one of the most popular being FSCS-ART [19]. With FSCS-ART, previously executed test inputs are stored in a set *T*, and whenever a new test input is needed, a fixed number of random inputs are generated as a *candidate set*, *C*, from which, based on some selection criteria, the best candidate is then chosen. One commonly used selection criterion is the *max–min* criterion, which involves selecting the candidate whose "smallest" distance to *T* is the *largest*. The FSCS-ART algorithm with *max–min* criterion is described in Procedure 1.

To use ART, some form of distance measurement is required, such as Euclidean distance for numeric inputs. For non-numeric inputs, the measurement should take account of the functionalities or characteristics of the SUT. Web applications, for example, often involve cookies, and Tappenden and Miller [31] proposed a cookie-based distance metric to evenly spread test inputs for web applications.

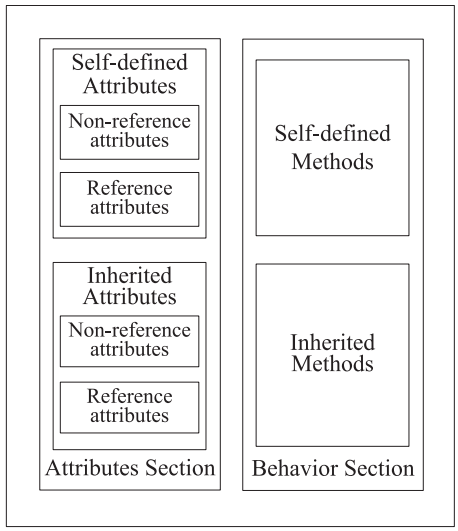

Fig. 1. Object structure.

C. Object and Test Input in OOS

There are three distinctive characteristics of OO programming: encapsulation, inheritance, and polymorphism. *Encapsulation* refers to keeping the object's attributes and implementation detail internally, but providing a simple interface (that is, a list of public constructors/methods) for other objects to interact with them. *Inheritance* means that one class (called a subclass) can be derived from other existing classes (called superclasses), and can also define its own unique properties and behavior. *Polymorphism* refers to the ability to present the same interface for entities of different types using techniques such as dynamic binding, method overloading, and method overriding [1].

The class is an abstraction of a set of objects with common attributes and methods, and is perhaps the most basic element of OOS. Various structures and relationships are built upon classes. For example, the inheritance relation is defined for classes as follows: when a class *A* inherits from another class *B*, an object of *A* will inherit elements from class *B*. When a class needs a continuous relation with another class, it will store an object reference of that class as one of its attributes.

Fig. 1 shows a typical object structure. Attributes and methods derived from the superclass are called inherited elements, while those defined within the class itself are called "self-defined" elements. Furthermore, attributes can be either *reference* or *nonreference*. A reference variable (abbreviated as *RefV*) refers to an object, while a nonreference variable (abbreviated as *NRefV*) stores values of primitive data types, such as *integers*, *real precision* numbers, *boolean* values, *characters*, *strings*, *enumerated* types, and *struct* values.

Classes provide semantic information for the OOS which can be very useful when designing test inputs. An OOS test input t often consists of two parts: $t.OBJ$ and $t.MINV$, where $t.OBJ$ is a list of objects and $t.MINV$ is an ordered list of methods

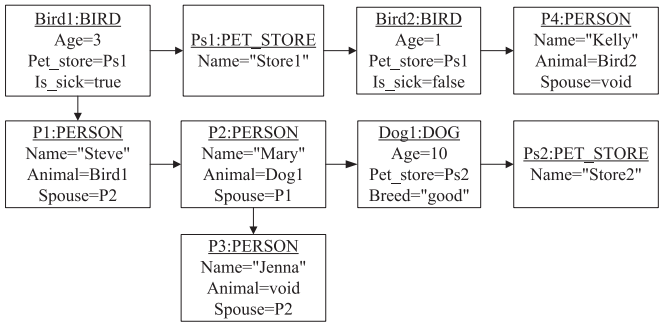

Fig. 2. Object diagram for the animal system from [16].

TABLE I
OBJECT DISTANCES (CALCULATED USING ARTOO) FOR ALL OBJECT PAIRS IN FIG. 2, AS REPORTED IN [16]

	Bird1	Bird2	Dog1	Ps1	Ps2	P1	P2	P3	P4
Bird1	0	1	13	N/A	N/A	N/A	N/A	N/A	N/A
Bird2	1	0	14	N/A	N/A	N/A	N/A	N/A	N/A
Dog1	13	14	0	N/A	N/A	N/A	N/A	N/A	N/A
Ps1	N/A	N/A	N/A	0	1	N/A	N/A	N/A	N/A
Ps2	N/A	N/A	N/A	1	0	N/A	N/A	N/A	N/A
P1	N/A	N/A	N/A	N/A	N/A	0	15.4	5.0	8.8
P2	N/A	N/A	N/A	N/A	N/A	15.4	0	16.0	15.0
P3	N/A	N/A	N/A	N/A	N/A	5.0	16.0	0	8.0
P4	N/A	N/A	N/A	N/A	N/A	8.8	15.0	8.0	0

(representing a sequence of method invocations) in the test input. An object used to test a nonstatic method (nsM) is known as the *method receiver* in OO programming, and is denoted $rec(nsM)$ in this paper. For any nonstatic method $t.minv_i \in t.MINV$, its receiver $rec(t.minv_i) \in t.OBJ$. Both $t.minv_i$ and $rec(t.minv_i)$ must refer to the same class. In general, $t = \{t.OBJ, t.MINV\}$, $size(t.OBJ) \geq 1$, and $size(t.MINV) \geq 1$. However, if we are only interested in the correctness of a constructor, then $size(t.MINV)$ will be 0; or if we are only interested in the correctness of a static method, then $size(t.OBJ)$ will be 0.

D. Distance Metric for ARTOO and Divergence-ART Approaches

ARTOO [16], [17] was proposed to test one specific method of a class, and hence the test input t_o for ARTOO consists of only one object $\{t_o.obj\}$ and one method $\{t_o.minv\}$, with $t_o.obj = rec(t_o.minv)$. Divergence-oriented ART [18] is an extension to ARTOO, and after an ARTOO test input t_o is constructed, $\{t_o.obj\}$ is used to call additional methods $\{t_d.minv_1, \ldots, t_d.minv_s\}$, which are invoked before $t_o.minv$. Hence, each test input t_d of divergence-oriented ART consists of one object $\{t_o.obj\}$ and one sequence of methods invocations $\{t_d.minv_1, \ldots, t_d.minv_s, t_o.minv\}$. Both ARTOO and divergence-oriented ART use the same distance metric, which, in the interest of clarity, in this paper is referred to as the "ARTOO" metric.

Ciupa *et al.* [16] used a PET_STORE example to explain ARTOO, and, for ease of comparison, their object diagram and distance calculation results are reproduced here in Fig. 2 and Table I, respectively. In Section III-D, we will also use the *PET_STORE* example to compare our proposed distance metric with the ARTOO metric, but because our distance metric takes account of method invocation, we expanded and revised their class diagram, as shown in Fig. 3.

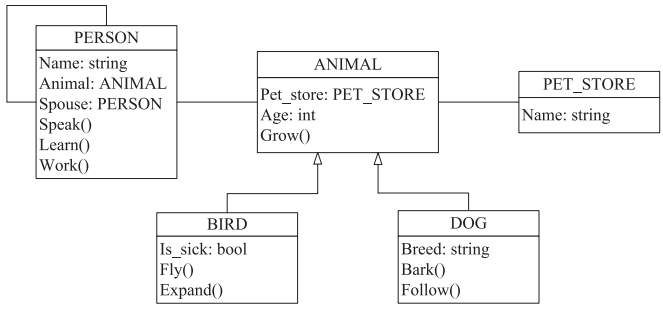

Fig. 3. Class diagram for an animal system with behavior (methods).

The following are three formulae used in the $ARTOO(p, q)$ metric [16], with p and q denoting any two objects of the same class, or derived from the same ancestor:

$$ARTOO(p, q) = field\_distance(p, q) + type\_distance(p, q) + recursive\_distance(p, q), \text{where}$$

$$field\_distance(p, q) = \phi * Avg(\Sigma_{a \in common(p.attribute,\, q.attribute)} weight_a * elementary\_distance(p.a, q.a)),$$

$$type\_distance(p, q) = \tau * \lambda * sum\_path\_length(p.type, q.type) + \tau * \nu * \Sigma_{a \in different(p.attribute,\, q.attribute)} weight_a$$

$$recursive\_distance(p, q) = \alpha * Avg(\Sigma_{a \in common(p.RefV,\, q.RefV)} weight_a * ARTOO(p.a, q.a))$$

The purpose of these three formulae are as follows.

1) The $field\_distance$ (p, q) formula calculates the distance between every two attributes $p.a$ and $q.a$ of the same type (*NRefV* or *RefV*), based on their value difference, i.e., $elementary\_distance(p.a, q.a)$. Standard ways exist to calculate the difference between two numbers, including, for example, $|p.a - q.a|$. When $p.a$ and $q.a$ do not belong to the same type, it is up to testers to define the formula for $elementary\_distance(p.a, q.a)$; for example, when $p.a$ and $q.a$ are reference variables, each of which stores objects, Ciupa *et al.* [16] proposed $elementary\_distance(p.a, q.a) = 0$ if $p.a$ and $q.a$ both either refer to the same object or to null; otherwise 10.
2) The $type\_distance$ (p, q) formula involves two calculations: a) summing the lengths of the paths from p and from q to their common superclass; and b) counting the variables of different types (i.e. variables left out by the $field\_distance$ calculation).

3) The $recursive\_distance\,(p, q)$ formula calculates the distance between two objects of the same type ($p.a$ and $q.a$) referenced in p and q, calling the $ARTOO(p.a, q.a)$ formula recursively. The depth of recursive calculation is taken into account, with the setting α (a proper fraction) used to attenuate the impact of deeper recursion.

In these formulae, Ciupa *et al.* set ϕ, τ, λ, ν, and $weight\_a$ to 1, and α to $1/2$ [16]. To illustrate how the ARTOO metric works, let us calculate the distance between $Bird1$ and $Dog1$ in Fig. 2.

According to Fig. 2, $Bird1$ and $Dog1$ have two common attributes: *age* (integer) and *Pet_store* (a reference variable of *PET_STORE*). Since $Ps1$ and $Ps2$ refer to different objects, their elementary distance is 10. Hence, $field\_distance(Bird1, Dog1)=1/2 * (|Bird1.age - Dog1.age| + 10)=1/2 * (7 + 10)=8.5$.

In addition, *DOG* and *BIRD* have one common superclass *ANIMAL*. The path length from $Dog1.type$ (*DOG* class) to *ANIMAL* and that from $Bird1.type$ (*BIRD* class) to *ANIMAL* are both 1. Hence, the sum_path_length($Dog1.type, Bird1.type$) is 2. Furthermore, $Bird1$ and $Dog1$ have two different attributes: *is_sick* (a boolean) and *breed* (a string), both of whose *weight_a* equal 1. Hence, $type\_distance(Bird1, Dog1)=2 + (1 + 1)= 4$.

Finally, $Bird1$ and $Dog1$ have one common reference variable, *Pet_store* (which is $Ps1$ and $Ps2$, respectively). The *recursive_distance* is equal to $\alpha * ARTOO(Ps1, Ps2)$. This will repeat the ARTOO distance calculation for $Bird1.Ps1$ and $Dog1.Ps2$. Because both $Ps1$ and $Ps2$ belong to the same class (*Pet_store*), their *type_distance* is zero. Since *Pet_store* does not have any reference variables, the *recursive_distance* between $Ps1$ and $Ps2$ is 0. The only distance between $Ps1$ and $Ps2$ is related to their common field: *name* (a string). Since $elementary\_distance(Ps1.\{Store1\}, Ps2.\{Store2\}) = 1$, we have $ARTOO(Ps1, Ps2)$ equal to 1. Consequently, $recursive\_distance(p, q) = \alpha * ARTOO(Ps1, Ps2) = (1/2) * 1 = 0.5$.

In summary, $ARTOO(Bird1, Dog1) = field\_distance(Bird1, Dog1) + type\_distance(Bird1, Dog1) + recursive\_distance(Bird1, Dog1) = 8.5 + 4 + 0.5 = 13$. If, on the other hand, we tried to apply $ARTOO(p.a, q.a)$ to measure the distance between $Bird1$ and $P1$, because they do not share the same superclass, their distance will be undefined, shown as N/A in Table I.

To illustrate how Ciupa *et al.* generate a test input using $ARTOO(p, q)$, suppose that $Fly(integer\ height, string\ place)$ is a method under test. To test this, $ARTOO(p, q)$ is used together with the FSCS-ART algorithm (Procedure 1) to separately generate an object of *BIRD* as the method receiver, as well as an "integer" and a "string" as arguments for the *Fly* method. These are then combined to form a test input, which can execute the Fly method. Throughout the entire test-input generation, three sets of executed objects (*T*) are maintained: one for *Bird*, one for *integers*, and one for *string* objects relevant to the *Fly* method. Clearly, it will be very complicated to measure the difference between test inputs, each of which consists of multiple objects and multiple methods of multiple classes.

III. Test Input and Object Distance

A. Current Distance Metric Issues

The discussion and examples in Section II-D highlight some issues for the current distance metrics.

1) OOS often consists of multiple classes, objects, and methods, which, for integration and system level testing, normally involves a list of objects and a sequence of method invocations in a test input. However, the distance metric used in ARTOO and divergence-oriented ART is not immediately suitable for the generation and even spreading of test inputs involving multiple objects and methods of multiple classes.
2) ARTOO was proposed to test OOS written in the Eiffel language, in which all objects have a *common* ancestor (a "super-class"). This was built into the ARTOO metric, allowing it to calculate the difference between two objects based on the distance to their closest common superclass. In C# and C++, however, objects do not necessarily have common superclasses, in which case, current distance metrics cannot show the object distance between them (such as for $Bird1$ and $P1$, in Table I, whose distance cannot be calculated, denoted as "N/A"). Because the ARTOO metric can only compute the distance between two objects if they belong to the same class or if they share a common ancestor, it is not suitable for calculating the difference between any two randomly chosen objects, whose classes are likely to be unrelated.
3) The $elementary\_distance(p.a, q.a)$ measures the distance between two attributes of the same type in p and q objects, but is currently restricted to numeric types, characters, strings, and tester-defined types associated with reference variables. The *enumerated*, *array*, *struct*, *union*, and *pointer* types, common features in C++ and C#, are not supported by current distance metrics.
4) Current distance metrics do not always return intuitive results. Although we might expect two objects of the same class to be closer than two objects of different classes (even if they share a common ancestor), some data in Table I violates this intuition: the distance between $Bird1$ and $Dog1$ (created from different classes) is 13, but the distance between $P2$ and $P3$ (both from Person class) is 16.
5) Current distance metrics do not work for test inputs involving more than one class, object or method invocation (such as those in Table II). Although divergence-oriented ART, as an enhancement to ARTOO, increases the number of methods per test input, these added methods are related to the same method receiver chosen by the ARTOO metric: neither ARTOO nor divergence-oriented ART consider interactions among multiple objects.

In order to address the above problems, we propose a new distance metric to calculate the distance between test inputs involving multiple objects and multiple method invocations of multiple classes. The calculation involves both the distance between objects (*OBJ*) and the distance between method

TABLE II
TEST INPUTS FOR THE ANIMAL SYSTEM IN FIG. 3

Test Input (T_1)	Test Input (T_2)	Test Input (T_3)
{	{	{
PET_STORE Ps1("Store1");	PET_STORE Ps1("Store1");	PET_STORE Ps2("Store2");
BIRD Bird1(3, &Ps1,true);	BIRD Bird2(1, &Ps1,false);	DOG Dog1(10, &Ps2,"good");
Bird1.Grow();	Bird2.Grow();	Dog1.Grow();
Bird1.Expand();	Bird2.Expand();	Dog1.Bark();
Bird1.Fly();	Bird2.Expand();	Dog1.Follow();
}	Bird2.Fly();	}
	}	

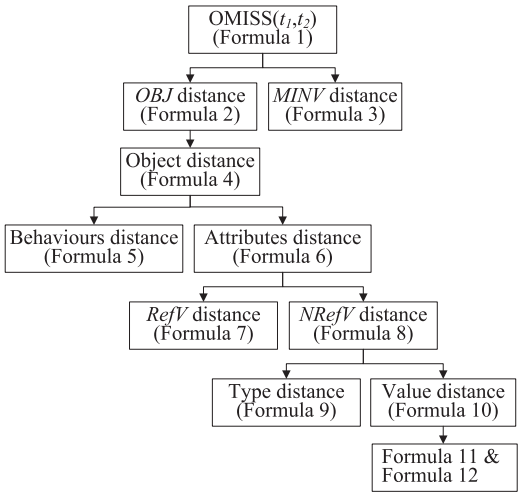

Fig. 4. Relationship among the 12 formulae.

invocation sequences (*MINV*) of two test inputs. Twelve formulae have been developed to calculate the distance between two test inputs t_1 and t_2, with Fig. 4 showing the relationship amongst these formulae. The formulae will be discussed in detail in Sections III-B and III-C.

The OMISS metric can also handle objects with both inherited elements and embedded objects (i.e., references to other objects), and complex data types such as *arrays*, *unions*, and *structs*. These data types can be used to store objects as well as primitive data, and which distance formula is used to calculate the distance between two arrays, unions or structs depends on the elements stored inside. Let us take the array as an example.

If arrays of "primitive type" are passed to a constructor, these arrays will be attributes inside an object, and Formula (12.8) will be used to calculate their difference. On the other hand, if the arrays are of a "nonprimitive type" (i.e., an array of references to objects) passed to a constructor, then they will be *RefV* attributes inside objects, and Formula (7) will be used to calculate the distance between them. If arrays of "nonprimitive type" are not passed to a constructor but used inside a test input, the objects referred to by these arrays will be the methods' receivers, and the distance between the two object sets will be calculated using Formula (2). Arrays that are not passed to a constructor, but are used as parameters for some method are not addressed in this paper.

B. Test Input Distance Formula

In this section, we explain how to calculate the distance between test inputs.

As discussed in Section II-C, a test input *t* for OO software consists of two parts, *t.OBJ* and *t.MINV*. Hence, the distance between two inputs, t_1 and t_2, should be the sum of their *t.OBJ* distance and *t.MINV* distance. The following formula shows the calculation:

$$TestcaseDistance(t_1, t_2) = TCobjDist(t_1.\textit{OBJ}, t_2.\textit{OBJ}) + TCmSeqDist(t_1.\textit{MINV}, t_2.\textit{MINV}). \quad (1)$$

The distance between two sets is generally determined by their elements, or more precisely, the distance between pairs of their elements. However, there are many ways to pair objects, resulting in different sums of distances. In this study, the minimum sum of distances amongst all possible ways of pairing is taken as the distance between t_1.*OBJ* and t_2.*OBJ* (explained in the next paragraph) as shown below:

$$TCobjDist(t_1.\textit{OBJ}, t_2.\textit{OBJ}) = \begin{cases} Min(\bigcup_{i=1}^{k!} \left\{ \sum_{j=1}^{k} ObjDist(t_1.obj_j, PL_i(t_2.\textit{OBJ}, j)) \right\}) \\ \quad \text{if } k = max(size(t_1.\textit{OBJ}), size(t_2.\textit{OBJ})) > 0 \\ 0 \quad \text{otherwise.} \end{cases} \quad (2)$$

The comparison of two sets of objects t_1.*OBJ* and t_2.*OBJ* using Formula (2) requires these sets to be of equal size—if one set is smaller, then it is augmented with an appropriate number of null-valued objects to increase its size. The distance between each pair of objects in the two sets is measured, and the sum of these distances (*DistSum*) is calculated. If there are k objects in the larger set, then there will be $k!$ different pair combinations and $k!$ possible *DistSum* values. Let us explain the notations in Formula (2) with an example. Suppose that $t_1.\textit{OBJ} = \{p1, p2\}$ and $t_2.\textit{OBJ} = \{q1, q2, q3\}$, then $k = 3$, $k! = 6$, and $t_1.\textit{OBJ}$ will become $\{p1, p2, null\}$. These six distinct combinations are shown in Fig. 5, in which there are six permuted lists of the object set $t_2.\textit{OBJ} = \{q1, q2, q3\}$. For ease of discussion, let $PL_2(t_2.\textit{OBJ})$ denote the second permuted list of $t_2.\textit{OBJ}$ in Fig. 5(b), and $DistSum(t_1.\textit{OBJ}, PL_2(t_2.\textit{OBJ}))$ denote the corresponding sum of distances between pairs ($p1$, $q1$), ($p2$, $q3$), and ($null$, $q2$) in $t_1.\textit{OBJ}$ and $PL_2(t_2.\textit{OBJ})$. In addition, let

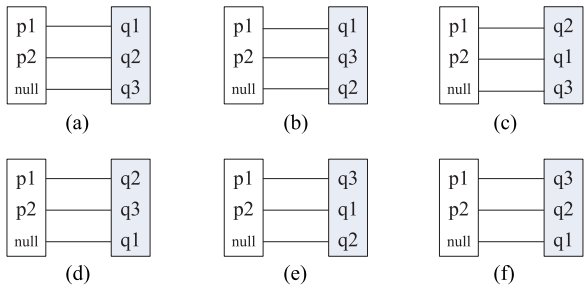

Fig. 5. All possible sums of distances for two object sets $t_1.OBJ$ and $t_2.OBJ$. (a) *DistSum*($t_1.OBJ$, $PL_1(t_2.OBJ)$), (b) *DistSum*($t_1.OBJ$, PL_2 ($t_2.OBJ$)), (c) *DistSum*($t_1.OBJ$,$PL_3(t_2.OBJ)$), (d) *DistSum*($t_1.OBJ$,$PL_4(t_2.OBJ)$), (e) *DistSum*($t_1.OBJ$,$PL_5(t_2.OBJ)$), and (f) *DistSum*($t_1.OBJ$,$PL_6(t_2.OBJ)$).

$PL_2(t_2.OBJ, 0)$, $PL_2(t_2.OBJ, 1)$, and $PL_2(t_2.OBJ, 2)$ denote elements $q1$, $q3$, and $q2$, respectively, in the second permuted list of $t_2.OBJ$ [see Fig. 5(b)]. These notations are used in Formula (2), according to which, the object distance (*TCobjDist*) between two object sets ($t_1.OBJ$ and $t_2.OBJ$) is the minimum *DistSum* between these two sets.

If neither set contains more than one object, then their distance is equal to the distance of their objects, calculated using Formula (4), that is, $TCobjDist(t_1.OBJ, t_2.OBJ) = ObjDist(t_1.obj_1, t_2.obj_1)$, where $t_1.obj_1$, and $t_2.obj_1$ are the single objects in the sets $t_1.OBJ$, and $t_2.OBJ$, respectively.

It should be noted that measuring the distance between $t_1.OBJ$ and $t_2.OBJ$ based only on the class structure will not work. Suppose that $size(t_1.OBJ) = size(t_2.OBJ) = 1$, and that $t_1.obj_1$ and $t_2.obj_1$ are both generated from the same class. If the measurement is only based on the class structure (attributes and methods defined inside a class), then the distance between these two objects will be zero; but there may be a significant difference in their attribute values. To address this, we suggest differentiating these two objects based on the constructor arguments used to initialize them. Such a distance definition is very similar to the elementary distance in ARTOO. However, we also suggest that this kind of distance should not be as significant as the distance based on the class structure, and so we propose a two-layer measurement: 1) the first measurement layer (the *primary distance*) is based on the internal structure of the classes from which the objects are created; and 2) the second measurement layer (the *secondary distance*) is based on the actual constructor arguments used to create the objects. Consequently, the distance between two object sets is presented in the form $a + b\mathbf{i}$, where a is the primary distance, b is the secondary distance, and $\mathbf{i}$ is the imaginary unit defined in this paper. Object distance calculation will be explained in Section III-C.

Unlike the distance measurement for object sets, there is only a one-layer measurement for method invocation (*MINV*) sets. Because the distance calculated from method invocation sets is as significant as the primary distance from object sets, we consider it the primary distance, and present it as $a + 0\mathbf{i}$. Every *MINV* set has three parts: length, method set, and invocation sequence. Hence, the distance between two *MINV* sets is the sum of the differences in lengths, in methods, and in invocation sequences. The following formula shows the detailed calculation:

$$\begin{aligned} TCmSeqDist(t_1.MINV, t_2.MINV) \\ &= LenD(t_1.MINV, t_2.MINV) \\ &+ MsD(t_1.MINV, t_2.MINV) \\ &+ SD(t_1.MINV, t_2.MINV) \end{aligned} \tag{3}$$

$$LenD(t_1.MINV,t_2.MINV) = |size(t_1.MINV) - size(t_2.MINV)| \tag{3.1}$$

$$MsD(t_1.MINV, t_2.MINV) = \begin{cases} 1 - \dfrac{size(t_1.MINV \cap t_2.MINV)}{size(t_1.MINV \cup t_2.MINV)}, \\ \qquad \text{if } min(size(t_1.MINV), size(t_2.MINV)) \geq 1 \\ 0 \qquad \text{otherwise} \end{cases} \tag{3.2}$$

$$SD(t_1.MINV, t_2.MINV) = \begin{cases} \dfrac{\sum_{i=1}^{n} d_i}{n}, \text{ where } d_i = \begin{cases} 1 & \text{if}(t_1.MINV[i] \neq t_2.MINV[i]) \\ 0 & \text{otherwise} \end{cases} \\ \qquad \text{if } n = min(size(t_1.MINV), size(t_2.MINV)) \geq 1 \\ 0 \qquad \text{otherwise} \end{cases} \tag{3.3}$$

In these formulae, $size(t_1.MINV)$ is defined as the length of the method invocation sequence in t_1. *LenD* denotes length difference, *MsD* denotes method set difference, and *SD* denotes sequence difference. In Formula (3.3), because the sequence differences are important, the calculation is based on the ordered lists, not sets. Comparing two methods involves the consideration of the following:

1) the method names;
2) the classes of the method receivers; and
3) the method signatures (the types and number of parameters)—the actual method arguments are not considered.

Given method invocation sequences $t_1.MINV = \{m_1, m_2, m_3\}$ and $t_2.MINV = \{m_3, m_2, m_1, m_4\}$, which have three common methods: according to Formula (3), the length difference is 1 ($= |3 - 4|$), the method difference is 0.25 ($= 1 - |\{m_1, m_2, m_3\}|/|\{m_3, m_2, m_1, m_4\}| = 1 - 3/4$), and the sequence difference is 0.667 ($= 2/3$), which gives a total difference (method invocation distance) of 1.917 ($= 1 + 0.25 + 0.667$).

Because of the two-layer measurement related to object sets, the test input distance also has primary and secondary distances, in the form of $a + b\mathbf{i}$.

There are two basic operations for the distance formulae:

Operation 1. *Addition*(+): For any two distances $s = a + b\mathbf{i}$ and $t = c + d\mathbf{i}$, $s + t = (a + b\mathbf{i}) + (c + d\mathbf{i}) = (a + c) + (b + d)\mathbf{i}$.

Operation 2. *Multiplication*($\times$). For any distances $s = a + b\mathbf{i}$, $s \times n = (a \times n) + (b \times n)\mathbf{i}$.

The following rule is used to compare any two distances $dist_1 = a + b\mathbf{i}$ and $dist_2 = c + d\mathbf{i}$.

$dist_1 = dist_2$ iff $(a = c)$ and $(b = d)$;

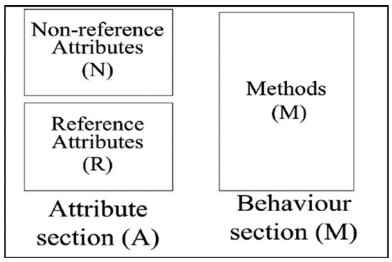

Fig. 6. Key object elements in our distance formula.

$dist_1 > dist_2$ iff $(a > c)$ or $(a = c$ and $b > d)$;
$dist_1 < dist_2$ iff $(a < c)$ or $(a = c$ and $b < d)$.

C. Object Distance Formula

In this section, we explain how to calculate the distance between two objects p and q. Since an object is created by calling a constructor of a class, inputs to our object distance formulae are the arguments passed to a constructor for initializing an object, as well as the object's internal elements, defined inside its class. The general object structure is shown in Fig. 1. However, because our distance formula does not differentiate between inherited and self-defined elements, we merge these and redraw the object structure as in Fig. 6, where $A = N \cup R$ denotes a list of *NRefV* and *RefV* attributes, and M denotes a list of methods inside an object.

Objects consist of a set of attributes and a set of methods (behavior), and hence the distance between two objects p and q is the sum of the distance between $p.M$ and $q.M$ and that between $p.A$ and $q.A$ [shown in Formula (4)]. The distance between $p.M$ and $q.M$ is determined by their method-sequence length and method set [see Formula (5)], and the distance between $p.A$ and $q.A$ is determined by their RefV attributes and NRefV attributes [see Formula (6)].

$$ObjDist(p, q) = \begin{cases} BehDist(p.M, q.M) + AttDist(p.A, q.A) \\ \qquad \text{if neither } p \text{ nor } q = null \\ 2 \qquad \text{if either } p \text{ or } q = null \\ 0 \qquad \text{if both } p \text{ and } q = null \end{cases} \tag{4}$$

Note: It is impossible for both *p* and *q* to be null in this study.

$$BehDist(p.M, q.M) = \begin{cases} |size(p.M) - size(q.M)| + (1 - \dfrac{size(p.M \cap q.M)}{size(p.M \cup q.M)}) \\ \qquad \text{if neither } p.M \text{ nor } q.M = null \\ 1 \qquad \text{if either } p.M \text{ or } q.M = null \\ 0 \qquad \text{if both } p.M \text{ and } q.M = null \end{cases} \tag{5}$$

$$AttDist(p.A, q.A) = \begin{cases} nonRefDist(p.A.N, q.A.N) + refDist(p.A.R, q.A.R) \\ \qquad \text{if neither } p.A \text{ nor } q.A = null \\ 1 \qquad \text{if either } p.A \text{ or } q.A = null \\ 0 \qquad \text{if both } p.A \text{ and } q.A = null. \end{cases} \tag{6}$$

If either $p.A.R$ or $q.A.R$ are nonempty sets (some *RefV* attributes exist for comparison between p and q), then the distance between them should be calculated using Formula (7). Since *RefV* attributes store objects, Formula (7) passes the stored objects to the *TCobjDist* distance metric in Formula (2) (which calculates the distance between two object sets). Formula (7) serves the same purpose as the *recursive_distance* formula in [16], and, to be consistent with their setup, we also set α to be $1/2$ to attenuate the impact of deeper recursive calculations. In the case where two objects mutually refer to each other, then the distance is set to 2.

$$refDist(p.A.R, q.A.R) = TCobjDist(p.A.R, q.A.R) * \alpha$$
$$\text{where } \alpha = \frac{1}{2}. \tag{7}$$

When there are *NRefV* attributes, the distance calculations are based on their data type and value difference, as shown in Formula (8), which produces the secondary distance.

$$nonRefDist(p.A.N, q.A.N) = typeDist(p.A.N, q.A.N) + (secDist(p.A.N, q.A.N))\mathbf{i} \tag{8}$$

where **i** is the imaginary unit.

Data type differences for the NRefV attributes include differences in size and set differences. Formulae (9), (9.1), and (9.2) show how to calculate the distance between the NRefV attributes of p and q, based on their data types (DT). If p and q are objects of the same class, their type distance, based on the *typeDist* metric, will be zero, which would obviously be insufficient for differentiating between the two objects. To address this, we define the *secDist* metric [see Formula (10)] to distinguish between two objects' NRefV attributes which are of the same data type.

$$typeDist(p.A.N, q.A.N) = TSizeDiff(p.A.N, q.A.N) + TSetDiff(p.A.N, q.A.N) \tag{9}$$

$$TSizeDiff(p.A.N, q.A.N) = |size(p.A.N.DT) - size(q.A.N.DT)| \tag{9.1}$$

$$TSetDiff(p.A.N, q.A.N) = size(p.A.N.DT \cup q.A.N.DT) - size(p.A.N.DT \cap q.A.N.DT). \tag{9.2}$$

Data types in C++ and C# can be classified into 11 categories:
1) *integer*;
2) *real number*;
3) *enumerated* types;

$$secDist(p.A.N,\, q.A.N) = \begin{cases} \sum dist(p.A.N[name_i],\, q.A.N[name_i]), & \text{if } p \text{ and } q \text{ are objects of the same class} \\ \sum_{i=1}^{11} gSecDist(pG[i],\, qG[i]), & \text{if } p \text{ and } q \text{ are objects of different classes} \end{cases} \tag{10}$$

$$gSecDist(pG[i],\, qG[i]) = Min\left(\bigcup_{s=1}^{m!}\left\{\sum_{j=1}^{m} dist(pG[i][j],\, PL_s(qG[i],\, j))\right\}\right), \tag{11}$$

where $m = max(size(pG[i]),\, size(qG[i])) > 0$.

$dist(x,\, y)$, where both x and y are of the same type : (12)

if x and y belong to one of ten commonly used types and either x or y is undefined (unknown value), then $dist(x,\, y) = 1$.

Note: It is impossible for both x and y to be undefined. (12.1)

If both x and y are defined as integer (int, short int, long int, signed int, or unsigned int) or real number (float or double) : (12.2)

$dist(x,\, y) = \dfrac{|x-y|}{Range}$, where $Range > 0$ and is a specific range of input domain.

If both x and y are defined as $enum$:

$$dist(x,\, y) = \begin{cases} \begin{cases} 0, & \text{if } x \text{ and } y \text{ are identical} \\ 1, & \text{otherwise} \end{cases} & \text{if the ordering of the values is not important} \\ \dfrac{|x-y|}{Range}, \text{ where } Range > 0 \text{ and is a specific range of input domain.} & \text{if the ordering of the values is important} \end{cases} \tag{12.3}$$

If both x and y are defined as $char$:

$$dist(x,\, y) = \frac{|ascii(x) - ascii(y)|}{Range} \tag{12.4}$$

where $ascii(x)$ represents the ASCII value of x, and $Range > 0$ and is a specific range of input domain.

If both x and y are defined as string :

$$dist(x,\, y) = \frac{editDistance(x,\, y)}{Range} \tag{12.5}$$

where $Range > 0$ and is a specific range of input domain or the length of the longest user-set input string.

If both x and y are defined as bool :

$$dist(x,\, y) = \begin{cases} 0, & \text{if } x \text{ and } y \text{ are identical} \\ 1, & \text{otherwise.} \end{cases} \tag{12.6}$$

If both x and y are defined as pointer of primitive type :

$$dist(x,\, y) = dist(*x,\, *y). \tag{12.7}$$

Note : $*x$ and $*y$ are the reference values of x and y, respectively.

If both x and y are defined as array or struct of primitive type, and $n = max(x.length, y.length) > 0$

$$dist(x,\, y) = \frac{\sum_{i=1}^{n} dist(x[i],\, y[i])}{n} \tag{12.8}$$

Note : if $max(x.length,\, y.length) = 0$, both x and y are undefined, but it will not occur in this study.

If both x and y are defined as union of primitive type:

$$dist(variable_{\in}x, variable_{\in}y) = \begin{cases} dist(variable_{\in}x,\, variable_{\in}y), & \text{if } variable_{\in}x \text{ and } variable_{\in}y \text{ belong to the same type} \\ 1, & \text{otherwise} \end{cases} \tag{12.9}$$

If x and y do not belong to ten commonly used types above, then $dist(x, y) = 0$. (12.10)

4) *character*;
5) *string*;
6) *Boolean*;
7) *pointer* to a primitive data;
8) *array* of primitive type;
9) *struct* of primitive type(s);
10) *union* of primitive type(s); and
11) "other" types.

The *secDist* formula [see Formula (10)] calculates the minimum sum of distances between every pair of *NRefV* attributes of the same primitive data type in $p.A.N$ and $q.A.N$ — $p.A.N[name_i]$ denotes the value associated with the variable $name_i$ in $p.A.N$. If p and q are objects of the same class, their sets of attribute names must be identical, and thus calculation of differences between *NRefV* attributes' is based strictly on their corresponding values; if p and q are objects of different classes, $p.A.N$ and $q.A.N$ need to be grouped according to the 11 categories of data types. Let $pG = \{pG[1], \ldots, pG[11]\}$ denote the 11 sets of *NRefV* attributes for $p.A.N$; and $qG = \{qG[1], \ldots, qG[11]\}$ for $q.A.N$, where $pG[i]$ and $qG[i]$ are groups of *NRefV* attributes of the same data type, DT_i, from $p.A.N$ and $q.A.N$, respectively.

In Formula (11), $pG[i][j]$ and $qG[i][j]$ denote the jth *NRefV* attributes of DT_i in the group $pG[i]$ and $qG[i]$, respectively. This formula requires pairing *NRefV* attributes in $pG[i]$ and $qG[i]$ such that it will return the minimum distance between these two sets. The procedure of pairing variables is similar to the procedure for pairing objects in $t_1.OBJ$ and $t_2.OBJ$ in Formula (2). Hence, if one group (either $pG[i]$ or $qG[i]$) has fewer variables than the other, then it is filled with supplemental, undefined values.

Formula (12) defines the distance calculation for two *NRefV* attributes x and y of the same type. Because a group of variables ($pG[i]$ or $qG[i]$) may contain undefined values (to make a smaller group be the same size as the larger one), Formula (12.1) handles this special case. Formulae (12.2) to (12.9) manage calculations for the ten commonly used data types in C++ and C#, and Formula (12.10) handles everything else, including "other" data types. The dynamic attributes in Formula (12), such as pointers and arrays, are arguments required by a constructor to create an object, or are required by a method under invocation. When calculating the distance for pointers and arrays in Formula (12), the initial values are used.

According to Formula (12.1), if either x or y is undefined, then the distance is set to 1. Formulae (12.2)–(12.5) all require a range, which, if not provided by the software specification, is assigned to be the default maximum and minimum values of the data type: for example, (signed) *integers* in C++ are between (-2^{32}) and $(2^{32} - 1)$. If the ordering is not important for an *enumerated* data type (e.g., CAR_MODEL = {Audi, BMW, Buick, Ford, Honda, Toyota, Nissan}), then the distance between x and y is 0 if they are identical, and 1, otherwise; if the ordering is important (e.g., WEEK_DAY ={Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}), then the distance is the difference between the position of x's value and that of y's value. The difference in ASCII code values is used for the distance calculation between two *characters*. Levenshtein distance [32] [namely the *editDistance* in Formula (12.5)] is used when calculating the difference between two *strings*: this is the minimum number of single-character edits (including insertion, deletion, and substitution) required to change one *string* into the other. If the range for two *strings* is not defined in the system specification, then it is set to the length of the longest user-set input *string*. The distance between two *Boolean* values is 1 if they are different, and 0, otherwise. For *pointers* of primitive types, the distance is calculated as the difference between the values pointed to. When the data in *arrays* and *structs* are of primitive types, the distance calculation for x and y uses the average distance between data pairs $x[i]$ and $y[i]$. If the *arrays* or *structs* have different sizes, then the smaller one is filled with enough *undefined* values to increase its size to that of the larger one. When x and y are *unions* (a special storage constructed to store a single data item, but which can be of more than one primitive type), if the variable types are different, then the distance is set to 1, otherwise the relevant distance metric from Formula (12) is selected to calculate the value difference.

D. Examples of Test Input Calculation and Object Distance Calculation

To illustrate how the formulae work, the different steps involved when calculating the *TestcaseDistance* between T_1 and T_3 in Table II are included in the Appendix. In this example, we assume that the length of the longest user-set string is 10, and that the input range of integer is from 0 to 100. As shown in the Appendix, the $Testcase Distance(T_1, T_3)$ is $4.267 + 0.22\mathbf{i}$. The same steps can be taken to calculate $Testcase Distance(T_1, T_2)$ and $Testcase Distance(T_2, T_3)$, which are $1.333 + 1.02\mathbf{i}$ and $5.267 + 0.24\mathbf{i}$, respectively.

This example illustrates two advantages of the proposed distance metric OMISS.

1) The metric not only considers the differences between test input objects, but also considers the differences between their method invocation sequences.
2) The metric can handle test inputs involving more than one object and class.

OMISS is also applicable when test inputs consist of only one object with no method invocations, or when they consist of only one static method with no object receiver.

Using the formulae in Section III-C, the distances for all object pairs shown in Fig. 2 can be calculated, as shown in Table III. The largest distance among all object pairs is between $Bird1$ (or $Bird2$) and $P1$, which is $7.5 + 0.25\mathbf{i}$; and the smallest distance is between $Ps1$ and $Ps2$, which is only $0.1\mathbf{i}$. Table III shows that the proposed metric can differentiate objects of different classes, returning a smaller value for objects of the same class and larger value for different classes.

E. Summary of OMISS Metric Advantages

A summary of the advantages of our proposed distance metric OMISS is as follows.

1) As discussed previously, the OMISS metric includes both object and test input distance metrics, and hence can measure the distance between any two test inputs consisting of a series of objects and a sequence of method invocations [see Formula (1)].

TABLE III
OBJECT DISTANCES (CALCULATED USING OMISS) FOR ALL OBJECT PAIRS IN FIG. 2

Objects	Bird1	Bird2	Dog1	Ps1	Ps2	P1	P2	P3	P4
Bird1	0	1.02i	2.8 + 0.12i	6	6	**7.5 + 0.25i**	7.5 + 0.15i	6.5 + 0.25i	7
Bird2	1.02i	0	2.8 + 0.14i	6	6	**7.5 + 0.25i**	7.5 + 0.15i	6.5 + 0.25i	7
Dog1	2.8 + 0.12i	2.8 + 0.14i	0	4 + 0.5i	4 + 0.5i	5.5 + 0.75i	5.5 + 0.55i	4.5 + 0.75i	5 + 0.5i
Ps1	6	6	4 + 0.5i	0	**0.1i**	3 + 0.3i	3 + 0.5i	2 + 0.6i	2 + 0.6i
Ps2	6	6	4 + 0.5i	**0.1i**	0	3 + 0.3i	3 + 0.5i	2 + 0.6i	2 + 0.6i
P1	**7.5 + 0.25i**	**7.5 + 0.25i**	5.5 + 0.75i	3 + 0.3i	3 + 0.3i	0	2.4 + 0.56i	1 + 0.5i	1 + 1.01i
P2	7.5 + 0.15i	7.5 + 0.15i	5.5 + 0.55i	3 + 0.5i	3 + 0.5i	2.4 + 0.56i	0	2.2 + 0.78i	2.4 + 0.47i
P3	6.5 + 0.25i	6.5 + 0.25i	4.5 + 0.75i	2 + 0.6i	2 + 0.6i	1 + 0.5i	2.2 + 0.78i	0	2 + 0.4i
P4	7	7	5 + 0.5i	2 + 0.6i	2 + 0.6i	1 + 1.01i	2.4 + 0.47i	2 + 0.4i	0

2) The OMISS metric can calculate the distance between two sets of multiple objects [see Formula (2)].
3) The OMISS metric can distinguish among method invocation sequences in test inputs [see Formula (3)].
4) The OMISS can measure distances between any two objects [see Formula (4)], without requiring that they are from the same class, or share a common ancestor.
5) The OMISS metric considers not only attribute distances, but also the behavioral distances among objects [see Formulae (5) and (6)].
6) The OMISS metric can differentiate between objects more precisely than ARTOO, even only based on the *type*, *field*, and *recursive* distances. It is because OMISS considers both the "recursive distance" [see Formula (7) for RefV attributes] and the "type distance" [see Formula (9)] as "primary distances," but the "field distance" as "secondary distance" [see Formula (10)]. This means that the internal structure of a class from which an object is built is considered more significant than the actual values used to initialize that object. With this two-layer measurement, OMISS can more easily distinguish between the origins of these objects than ARTOO can, by reducing the influence of their individual differences (attribute values).
7) The type, reference, and elementary distances are not treated equally by OMISS metric, i.e., the elementary distance is considered secondary [see Formulae (8) and (10)].

IV. DEVELOPMENT OF THE OMISS-ART TOOL

As discussed in the previous section, OMISS has many advantages. Using this metric with the FSCS-ART algorithm (Procedure 1) can better evenly spread test inputs for integration testing, however, two critical steps need to be completed before applying it. Sections IV-A and IV-B explain these two steps, and Section IV-C explains how OMISS and FSCS-ART are implemented with a forgetting strategy to reduce the distance computation overheads.

A. Obtaining the Class Diagram Information Through Source Code Analysis

Because OOS is centered around classes, which carry very useful information for the creation of test inputs and distance measurements, the first step involves obtaining the class information (e.g., the class diagram) from the subject program's source code. Many tools exist which can facilitate this, including Doxygen [33] and Understand [34]; we used Visual Studio 2010, which is an integrated development environment from Microsoft [35]. We developed a tool, OMISS-ART, which uses the class diagram to extract class information, such as the attributes list, and methods' details.

B. Generating a Pool of Valid Test Inputs

After obtaining the class information, OMISS-ART can create random test inputs, each of which has a set of objects and a set of methods invoked in a sequence. However, many randomly constructed inputs contain invalid method invocation sequences, which may lead to abnormal termination of the program. In this study, therefore, we excluded the invalid sequences to create a pool of valid test inputs, as shown in Procedures 2–4.

The *TestInputsGen* function (Procedure 2) makes use of the class information to generate a specified number of test inputs (*TInputNum*), calling *OBJGen* (Procedure 3) and *MINVGen* (Procedure 4) to generate the objects (*t.OBJ*) and methods invocation sequence (*t.MINV*), respectively. The test input pool is created by combining each *t.OBJ* and *t.MINV* to form a test input *t*, discarding any input that causes the program to terminate abnormally. We use this approach to generate a test input pool for each subject program in the empirical study (see Section V), thus ensuring that all evaluated RT approaches can select valid inputs during testing.

C. FSCS-ART With the Random Forgetting Strategy

After completing the above steps, OMISS can be used with the FSCS-ART algorithm (Procedure 1) to determine which candidate test case from the pool is furthest away from previously executed tests. In OMISS-ART, once the algorithm identifies the best candidate from ten randomly constructed inputs in the candidate set *C*, this candidate is then used to test the subject program and added to the executed test set *T*. To reduce some of the $O(n^2)$ computational overheads associated with FSCS-ART (Steps 8–20 in Procedure 1), three *forgetting strategies* [41] (random forgetting; consecutive retention; and restarting) have been proposed to "forget" some previously executed test inputs (Step 10). In this study, the random forgetting strategy was implemented to reduce the overheads by randomly choosing a fixed number ($h = 20$ in OMISS-ART) of items from *T*, and only using these in distance calculations to decide the best candidate in *C*.

Procedure 2: Generating test inputs.

TestInputsGen($classInfo$, $TInputNum$, $MaxLengthMINV$, $MaxLengthOBJ$)
Set TestPool $= \Phi$;
for i =1 to $TInputNum$ **do**
  $n =$ a random number between 1 and $MaxLengthMINV$
  $m =$ a random number between 1 and $MaxLengthOBJ$
  $OS =$ **OBJGen**($classInfo$, m);
  Set $MList = \Phi$;
  for each object obj in OS **do**
    $MList = MList \cup \{obj's$ public methods given in $classInfo\}$
  end for
  Set $invalidTC = true$;
  while $invalidTC$ **do**
    $MethodSeq =$ **MINVGen**(MList, n);
    Create a test input t_i by combing OS and $MethodSeq$;
    Run the program with t_i
    if t_i causes abnormal program termination **then**
      set $invalidTC = true$
      else set $invalidTC = false$
    end if
  end while
  $TestPool = TestPool \cup \{t_i\}$
end for
return $TestPool$

Procedure 3: Generating objects.

OBJGen($classInfo$, m)
Set $ObjectSet = \Phi$;
Set $i = 0$;
while $i \neq m$ **do**
  Randomly select a class ***Cls*** from classInfo
  Randomly select a constructor from ***Cls***, and randomly generate arguments for it
  Construct an object o_i using the chosen constructor and generated arguments
  if o_i does not cause abnormal program termination **then**
    $ObjectSet = ObjectSet \cup \{o_i\}$;
    $i = i + 1$;
  end if
end while
return $ObjectSet$;

Procedure 4: Generating a method invocation sequence.

MINVGen($MLis$, n)
Set $MethodSeq = \Phi$;
for i =1 to n **do**
  Randomly select a method md_i from ***MList***, and generate arguments for md_i
  $MethodSeq = MethodSeq \cup \{md_i\}$;
end for
return $MethodSeq$;

In summary, the testing procedure in OMISS-ART consists of three steps:

1) analyzing the subject program and obtaining class information;
2) generating valid random test inputs; and
3) selecting test inputs for the SUT using the FSCS-ART algorithm with the random forgetting strategy.

V. Empirical Studies and Analysis

In this section, we describe two series of empirical studies comparing the performance of OMISS-ART with existing RT approaches.

A. Setup of the Empirical Studies

In the software testing community, mutated programs are often used when studying the failure-detection effectiveness of a testing approach [42]. After seeding a fault into a subject program, if the subject program and a faulty version (a mutant) produce different outputs for a given test input, then the mutant is said to be killed, meaning that the test input detects a failure. In this study, we also used mutation testing to study the effectiveness and efficiency of OMISS-ART.

In our empirical studies, the subject programs involved a sequence of operations (methods) to achieve certain tasks. Such programs are known as *interactive programs* [43], [44], and have both method and object interactions. The 17 programs in this study, all written in the C++ or C# language, are from open sources [36]–[40]. Methods were randomly selected from each program, and a single fault seeded into the method according to one of the following 13 mutation operators [42], [45]. Of these 13 mutation operators, the last 6 are OO-specific, and are used to generate OO-specific faults.

1) arithmetic operators replacement (AOR);
2) logical operators replacement (LOR);
3) relational operators replacement (ROR);
4) constant for scalar variable replacement (CSR);
5) scalar variable for scalar variable replacement (SVR);
6) scalar variable for constant replacement (SCR);
7) array reference for constant replacement (ACR);
8) new method invocation with child class type (NMI);
9) argument order change (AOC);
10) accessor method change (AMeC);
11) access modifier change (AMoC);
12) hiding variable deletion (HVD);
13) property replacement with member field (PRM).

Table IV shows the details for the subject programs and their seeded faults, and Table V shows the type of mutation operators and the number of faults seeded for each program.

TABLE IV
SUBJECT PROGRAMS

SUT ID	SUT name	Lines of code	Num. of public classes	Num. of public methods	Num. of faults	Description
1	CCoinBox [36]	120	1	7	4	C++ library that simulates a *vending machine*
2	Calendar [36]	287	5	27	5	C++ library for *calendar* operation
3	Stack [37]	420	3	13	5	Microsoft C# library for *stack* operation
4	Queue [36]	201	3	12	4	Microsoft C# library for *queue* operation
5	WindShieldWiper [36]	233	1	13	4	C++ library that simulates a *windshield wiper*
6	SATM [36]	197	1	9	4	C++ library that simulates an *ATM*
7	BinarySearchTree [37]	588	3	19	7	C# library for binary search tree algorithms
8	RabbitAndFox [37]	770	6	33	9	C# program that simulates a predator-prey model
9	WaveletLibrary [38]	2406	12	84	15	C# library for wavelet algorithms
10	BackTrack [37]	1051	9	27	13	C# library for backtracking algorithms
11	NSort [39]	1118	18	61	14	C# library for sorting algorithms
12	SchoolManagement [37]	1726	11	131	21	C# program for managing school activities
13	EnterpriseManagement [37]	1357	8	76	8	C# program for managing enterprise business
14	ID3Manage [37]	4538	28	129	12	C# library for reading and writing of ID3 tags in MP3 files
15	IceChat [38]	71000	101	271	24	C# program that implements an IRC (Internet Relay Chat) Client
16	CSPspEmu [40]	406808	443	1433	26	C# program for a PSP (PlayStation Portable) emulator
17	poco-1.4.4: Foundation [40]	149547	641	4480	28	C++ library that contains a platform abstraction layer and a large number of useful utility classes

TABLE V
MUTATION OPERATORS AND THE NUMBER OF FAULTS SEEDED

SUT ID	Total number of faults	General mutation operators (number)	OO mutation operators (number)
#1	4	AOR(1), LOR(2), ROR(1)	N/A
#2	5	AOR(1), LOR(1), ROR(2), CSR(1)	N/A
#3	5	AOR(1), LOR(1), ROR(1), SVR(2)	N/A
#4	4	AOR(1), LOR(1), ROR(1), SCR(1)	N/A
#5	4	AOR(1), LOR(1), ROR(1), ACR(1)	N/A
#6	4	AOR(1), LOR(1), ROR(1), SCR(1)	N/A
#7	7	AOR(1), LOR(1), SVR(2)	NMI(1),AOC(1), AMeC(1)
#8	9	AOR(1), LOR(1), SVR(1)	NMI(1),AOC(1), AMeC(1),AMoC(1),HVD(1),PRM(1)
#9	15	LOR(1), SVR(1),CSR(2), SCR(1), ACR(2)	NMI(1),AOC(1), AMeC(1), AMoC(2), HVD(2), PRM(1)
#10	13	AOR(1), ROR(1),CSR(1), SCR(2), ACR(1)	NMI(2), AOC(2), AMeC(1), AMoC(1), PRM(1)
#11	14	AOR(1), LOR(1), ROR(2), SCR(2), ACR(1)	NMI(2),AOC(2), AMeC(1), AMoC(2)
#12	21	AOR(2), LOR(3), ROR(1), SVR(2),CSR(2), SCR(1), ACR(2)	NMI(1), AOC(1), AMeC(3), AMoC(1), HVD(1), PRM(1)
#13	8	ROR(1),CSR(1), SCR(1)	NMI(2), AOC(2), AMeC(1)
#14	12	AOR(1), CSR(1), ACR(2)	NMI(1), AOC(2), AMeC(2) AMoC(2), PRM(1)
#15	24	AOR(2), LOR(1), ROR(1), SVR(2),CSR(2), SCR(1),ACR(2)	NMI(2), AOC(3), AMeC(2), AMoC(2), HVD(2), PRM(2)
#16	26	AOR(2), LOR(1), ROR(1),SVR(2),CSR(1), SCR(1),ACR(2)	NMI(2), AOC(3), AMeC(3),AMoC(3), HVD(3), PRM(2)
#17	28	AOR(1), LOR(2), ROR(2), SVR(1),CSR(2),SCR(1),ACR(1)	NMI(2), AOC(3), AMeC(4), AMoC(3), HVD(3), PRM(3)

As with other ART studies, the *F*-measure (the number of test inputs required to detect the first failure, F_m) and the *E*-measure (the expected number of failures detected, E_m) [15] are used in this study to measure the effectiveness of the proposed approach. The time taken to detect the first failure, F_m-$time$, was also recorded. The machine used to conduct testing has an Intel dual core i3-2120 3.3 GHz processor, 4 GB of RAM, and runs under the Windows 7 operating system.

OMISS-ART is a tool we developed using the Microsoft .NET framework to implement the proposed approach. OMISS-ART can test C++ and C# programs. In the study, our proposed approach was compared with four other testing approaches:

1) ARTOO;
2) Divergence-oriented ART (DO-ART);
3) RT with one method invocation (RT-1); and
4) RT with multiple object interactions and method invocations (RT-n).

Both ARTOO and divergence-oriented ART were redeveloped in .NET to facilitate their application to C++ and C# programs. The redeveloped versions were extensively tested and verified, including by using the original examples in the ARTOO and divergence-oriented ART literature. Table VII summarizes the structures of test inputs generated by the different approaches, with the column "Evenly spread" denoting whether or not methods and objects in the test inputs have been evenly spread.

To use OMISS-ART in this study, the three parameters *TInputNum*, *MaxLengthMINV*, and *MaxLengthOBJ* for Procedure 2 in Section IV were set to 10000, 5, and 5, respectively. When calculating object distances with the ARTOO metric for ARTOO and DO-ART, if a program has primitive data types (such as *enum*, *array*, *struct*, and *union*) which are not handled by the ARTOO metric, then the elementary distance was arbitrarily set to zero. The same settings used in divergence-oriented

TABLE VI
F_M AND F_M-TIME COMPARISON

SUT ID	F_m					F_m-time (Seconds)				
	OMISS-ART	ARTOO	DO-ART	RT-1	RT-n	OMISS-ART	ARTOO	DO-ART	RT-1	RT-n
#1	70.17	128.87	103.05	194.00	71.94	1.32	0.25	0.30	0.16	0.54
#2	2.47	10.11	4.53	10.47	3.10	1.18	0.40	0.55	0.38	1.04
#3	19.47	42.72	15.36	44.27	24.76	2.68	0.81	1.33	0.81	1.64
#4	6.58	14.06	6.59	14.69	8.79	1.64	0.77	1.68	0.75	1.55
#5	66.51	173.59	80.13	197.69	70.09	1.06	0.25	0.33	0.19	0.62
#6	46.00	152.29	39.76	121.95	53.54	1.32	0.53	0.63	0.31	0.68
#7	27.24	90.75	40.29	123.42	31.65	4.49	0.87	1.52	0.85	2.32
#8	20.48	68.47	30.13	86.40	23.75	3.09	0.69	0.81	0.68	1.68
#9	6.90	22.26	10.83	24.72	7.71	3.52	1.43	1.94	1.39	3.40
#10	2.39	5.37	2.97	5.19	2.67	0.96	0.39	0.60	0.42	0.95
#11	32.84	91.91	76.59	82.51	40.48	1.11	0.18	0.24	0.16	0.49
#12	55.92	253.07	93.79	275.40	71.73	3.48	0.61	1.06	0.57	1.66
#13	37.80	123.03	68.94	154.28	45.53	3.57	1.12	1.13	1.07	1.98
#14	31.36	150.46	56.09	106.59	39.35	1.58	0.25	0.38	0.20	0.82
#15	97.80	341.73	252.38	398.12	147.36	3.44	0.99	1.31	0.61	2.72
#16	133.38	369.96	248.24	386.49	192.89	4.40	1.84	2.36	1.63	3.51
#17	180.28	377.58	257.30	410.54	212.37	4.82	2.20	2.87	1.97	3.99
mean	49.27	142.13	81.59	155.10	61.63	2.57	0.80	1.12	0.71	1.74
sDev	68.27	182.96	117.33	205.79	85.76	2.86	0.76	1.02	0.66	1.44
error (95% CL)	±2.06	±5.53	±3.55	±6.22	±2.59	±0.09	±0.02	±0.03	±0.02	±0.04
uppB	51.33	147.67	85.14	161.33	64.22	2.66	0.82	1.15	0.73	1.78
lowB	47.21	136.60	78.04	148.88	59.04	2.48	0.78	1.09	0.69	1.70
Interval where 95% data fall	4.13	11.07	7.10	12.45	5.19	0.17	0.05	0.06	0.04	0.09
Accuracy%	±4.19	±3.89	±4.35	±4.01	±4.21	±3.36	±2.86	±2.74	±2.78	±2.50
max	484.00	1283.00	1084.00	2509.00	537.00	35.53	8.51	14.67	3.78	8.75
min	1.00	1.00	1.00	1.00	1.00	0.01	0.01	0.02	0.01	0.02
range	483.00	1282.00	1083.00	2508.00	536.00	35.53	8.50	14.65	3.77	8.74

TABLE VII
TEST INPUT CHARACTERISTICS (M: METHODS, O: OBJECTS)

Testing approach	Number of objects in t.***OBJ***	Number of classes used to create t.***OBJ***	Number of methods in t.***MINV***	Evenly spread
OMISS-ART	≥ 1	≥ 1	≥ 1	M and O
ARTOO	1	1	1	O
DO-ART	1	1	≥ 1	O
RT-1	1	1	1	N/A
RT-n	≥ 1	≥ 1	≥ 1	N/A

ART [18] were used in this study: *create-deep* was set to 4; and both *max-call* and *max-diversify* were set to 5. The *create-deep* setting is the depth of recursively created reference objects; *max-call* is the maximum number of calls to the same public method of the same object; and *max-diversify* is the maximum number of calls to different public methods in the same object. Furthermore, similar to the ARTOO and divergence-oriented ART studies, some special numeric values were also considered when generating test inputs:

1) MAX_VALUE;
2) MIN_VALUE;
3) MAX_VALUE − 1;
4) MIN_VALUE + 1;
5) −1;
6) 0; and
7) 1.

Each special value had a probability of $0.25/7$ of being selected for testing. The $null$ value for variables of object types was not considered as a special value in the experiment because the probability of its occurrence could not be obtained from the ARTOO or divergence-oriented ART studies. To ensure that these experiments were not interrupted by program crashes caused by invalid inputs, Procedure 2 was used to generate the pools of valid test inputs for each of the programs.

B. Experiments

1) Experiment I to Measure F_m and F_m-Time: Table VI summarizes the F_m and F_m-$time$ results. All results in this table are averaged over 300 runs of tests for each subject program using different seeds. The statistical data in Table VI (from "mean" to "range") are based on 5100 ($300 * 17$) datasets. In the experiment, the confidence level was set to 95%, giving the sample means, standard deviation (*sDev*), accuracy, and confidence intervals (*lowB* and *uppB*), as shown in Table VI.

Table VI shows that OMISS-ART outperforms other approaches in terms of F_m, except for two cases (Programs #3 and #6, where DO-ART outperforms OMISS-ART); but spends more time to find the first failure. This is because calculating the test input distances when deciding the best candidate is time consuming. In most cases (14 out of 17), F_m-$time$ for OMISS-ART is not more than twice that of RT-n, which was found to be the best RT algorithm for OOS testing in this study. From the statistical data of Table VI, we have the following observations related to F_m.

TABLE VIII
STATISTICAL RESULTS OF F_M FOR 17 SUBJECT PROGRAMS

SUT ID		OMISS-ART	ARTOO	DO-ART	RT-1	RT-n
#1	mean	70.17	128.87	103.05	194.00	71.94
	sDev	61.14	104.33	93.31	199.55	69.36
#2	mean	2.47	10.11	4.53	10.47	3.10
	sDev	1.72	9.31	3.85	9.16	2.82
#3	mean	19.47	42.72	15.36	44.27	24.76
	sDev	20.73	41.18	16.46	43.22	23.90
#4	mean	6.58	14.06	6.59	14.69	8.79
	sDev	5.63	12.88	5.45	13.34	8.90
#5	mean	66.51	173.59	80.13	197.69	70.09
	sDev	70.37	144.83	81.72	195.15	64.54
#6	mean	46.00	152.29	39.76	121.95	53.54
	sDev	49.94	149.56	39.12	116.92	51.69
#7	mean	27.24	90.75	40.29	123.42	31.65
	sDev	28.50	70.54	35.39	119.61	31.22
#8	mean	20.48	68.47	30.13	86.40	23.75
	sDev	19.66	69.35	32.56	81.07	24.97
#9	mean	6.90	22.26	10.83	24.72	7.71
	sDev	6.39	21.25	9.59	24.55	6.67
#10	mean	2.39	5.37	2.97	5.19	2.67
	sDev	1.75	4.69	2.32	4.61	1.87
#11	mean	32.84	91.91	76.59	82.51	40.48
	sDev	47.56	142.19	124.10	134.37	64.92
#12	mean	55.92	253.07	93.79	275.40	71.73
	sDev	64.71	242.72	90.62	302.88	71.56
#13	mean	37.80	123.03	68.94	154.28	45.53
	sDev	37.99	123.57	66.63	150.86	41.53
#14	$mean$	31.36	150.46	56.09	106.59	39.35
	sDev	31.98	156.40	69.39	97.89	43.29
#15	mean	97.80	341.73	252.38	398.12	147.36
	sDev	57.44	196.96	140.09	231.05	84.30
#16	mean	133.38	369.96	248.24	386.49	192.89
	sDev	81.00	221.48	127.74	219.80	105.30
#17	mean	180.28	377.58	257.30	410.54	212.37
	sDev	96.84	219.80	152.94	244.35	122.36

1) The F_m was accurate to within $\pm 5\%$ of the mean value, with 95% confidence.
2) The confidence intervals between the lower and upper bounds (*lowB* and *uppB*) do not overlap on F_m.
3) The standard deviation for OMISS-ART is the smallest (68.27).
4) There is no overlap between the $lowB$ and $uppB$ for F_m-$time$.

Based on these observations, we can conclude that the F_m and F_m-$time$ experimental results statistically have good reliability [46], which implies that the *F*-measure for OMISS-ART is lower than those of other approaches, with high probability.

In order to further analyze the F_m of each testing approach for each subject program, Table VIII summarizes the main statistical measures ($mean$ and $sDev$) for the 17 subject programs. Figs. 7– 23 show the experimental results in box plots for each subject program. From these statistical data, we have the following observations.

1) OMISS-ART outperforms other approaches in terms of the mean values of F_m, except for DO-ART in Programs #3 and #6.
2) In most cases (13 out of 17), the standard deviation for OMISS-ART is the smallest (not for Programs #3, #4, #5, and #6).

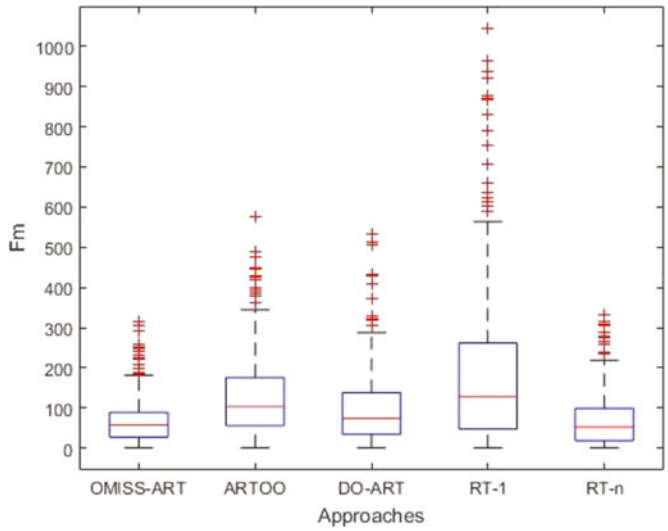

Fig. 7. Experimental result of F_m for Program #1.

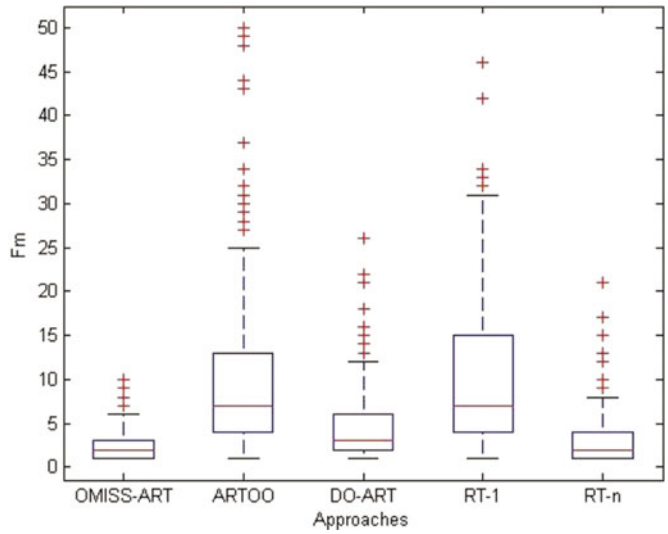

Fig. 8. Experimental result of F_m for Program #2.

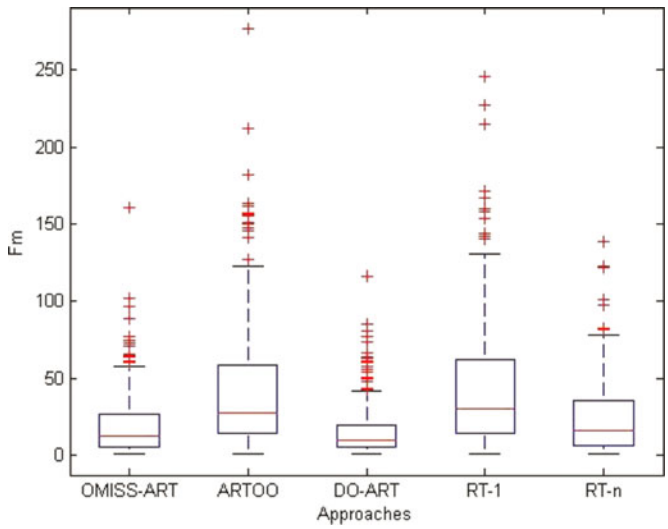

Fig. 9. Experimental result of F_m for Program #3.

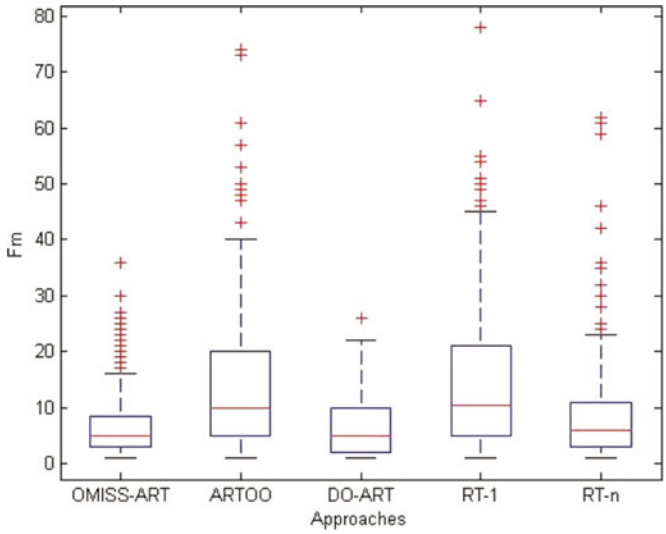

Fig. 10. Experimental result of F_m for Program #4.

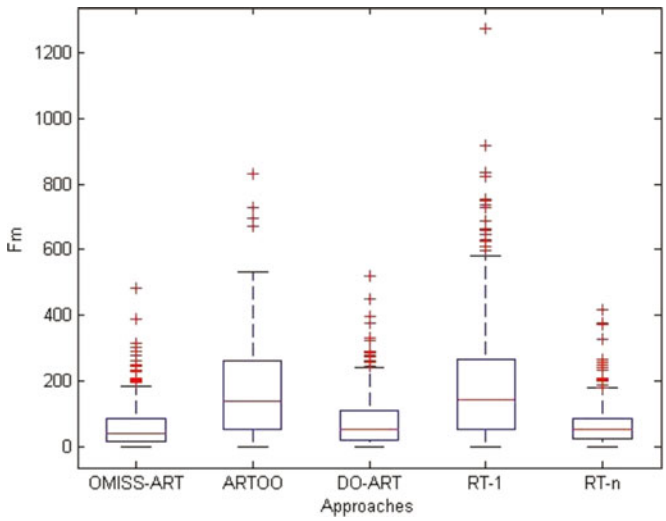

Fig. 11. Experimental result of F_m for Program #5.

3) For some larger-scale programs (Programs #9, #14, #15, #16, and #17), OMISS-ART clearly performs better than other testing approaches.
4) In most cases (except for Programs #3 and #6), the interquartile range (the box length) and maximum (the top end of the whiskers) of OMISS-ART are smaller than those of other approaches, which shows that OMISS-ART has more stable data distribution than other testing approaches.

To facilitate the data comparison, we calculated the ratio between F_m of OMISS-ART and that of the other approaches, reporting the minimum, maximum, and average ratios in Table IX. Table IX shows that the majority of F_m ratios are less than 1, except for two cases (Programs #3 and #6, where DO-ART outperforms OMISS-ART). For detection of the first failure, OMISS-ART required an average of 35%, 71%, 33%, and 83% of the number of test inputs of ARTOO, DO-ART, RT-1, and RT-n, respectively. Furthermore, RT-1 took the least

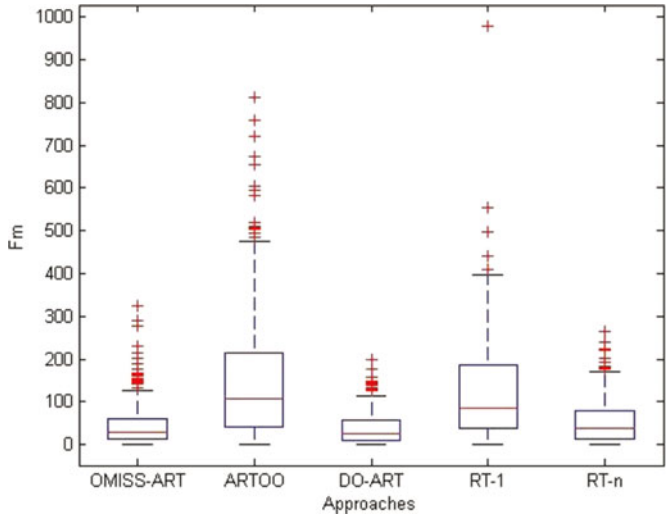

Fig. 12. Experimental result of F_m for Program #6.

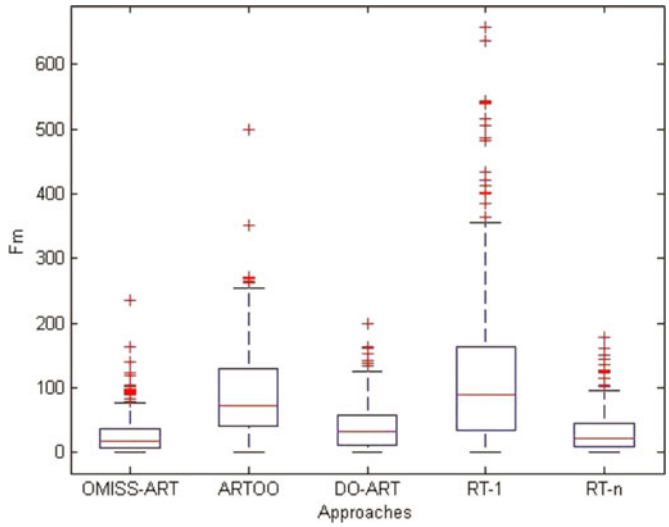

Fig. 13. Experimental result of F_m for Program #7.

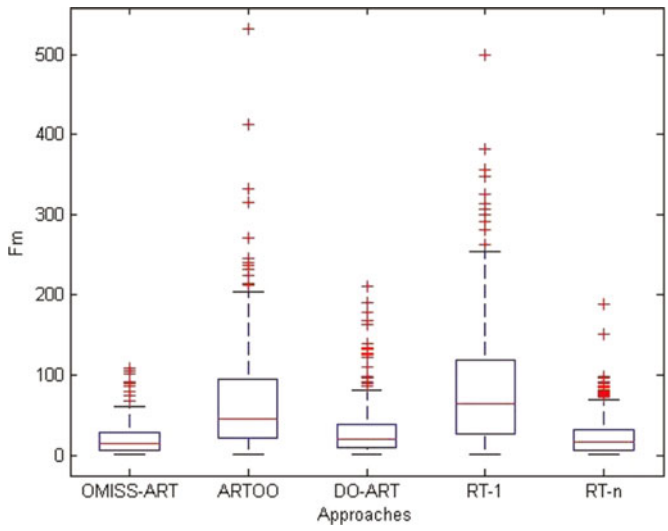

Fig. 14. Experimental result of F_m for Program #8.

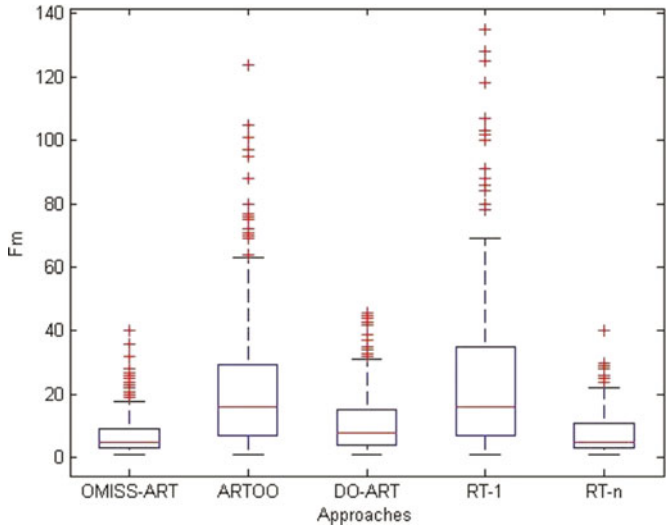

Fig. 15. Experimental result of F_m for Program #9.

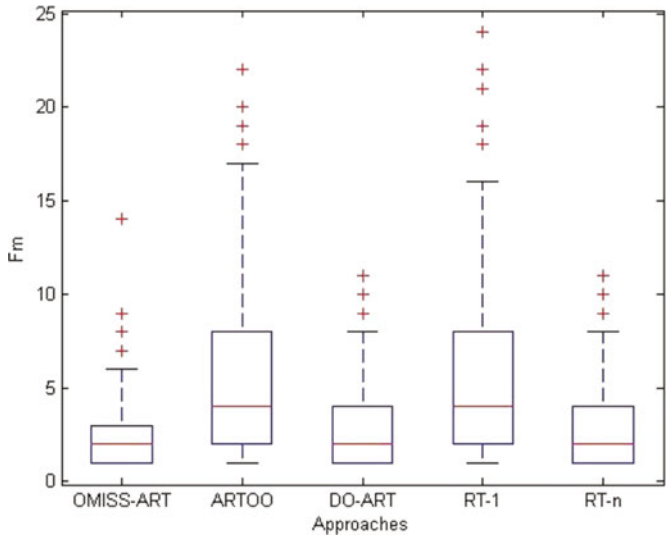

Fig. 16. Experimental result of F_m for Program #10.

time to find the first failure, followed by ARTOO, DO-ART, RT-n, and OMISS-ART.

Another finding was that RT-n is generally better than ARTOO, DO-ART, and RT-1, the reason being that RT-n uses multiple classes to generate objects in *OBJ* and method calls in *MINV*, for each test input t. As shown in Table VII, the structure of those inputs generated by RT-1 and ARTOO are similar, and so are those created by RT-n and OMISS-ART: the differences between them is the presence or absence of distance measurement and best candidate selection. This empirical study shows that involving multiple classes, objects, and methods may be the key to improving failure detection in OOS testing.

The cases where ARTOO performs similarly to RT-1 according to respective F_m are related to those subject programs using test inputs with data types, such as *enum*, *array*, *struct,* and *union*, which were not differentiated by the ARTOO distance metric for even spreading. However, ARTOO outperforms

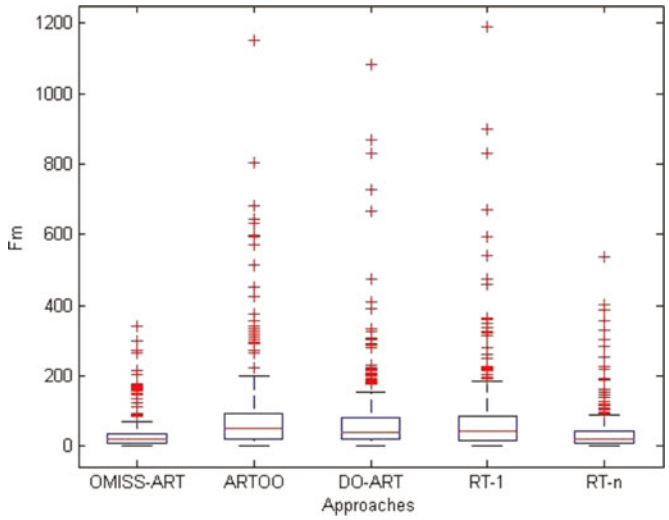

Fig. 17. Experimental result of F_m for Program #11.

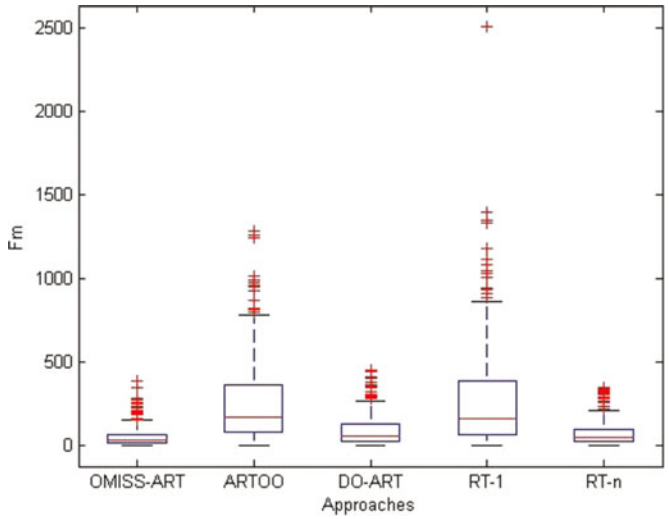

Fig. 18. Experimental result of F_m for Program #12.

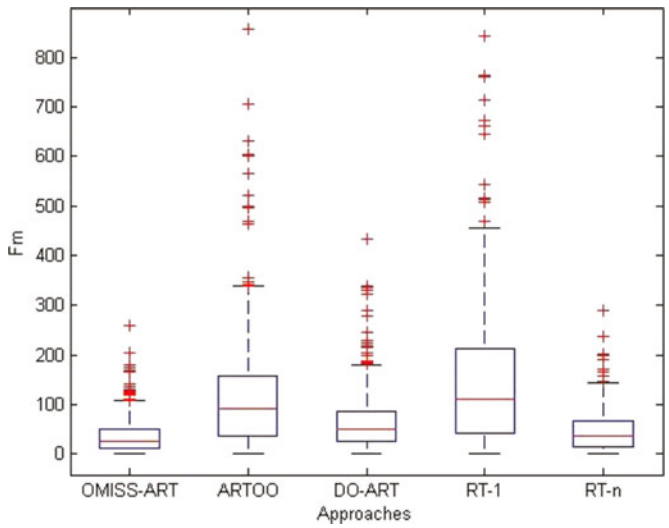

Fig. 19. Experimental result of F_m for Program #13.

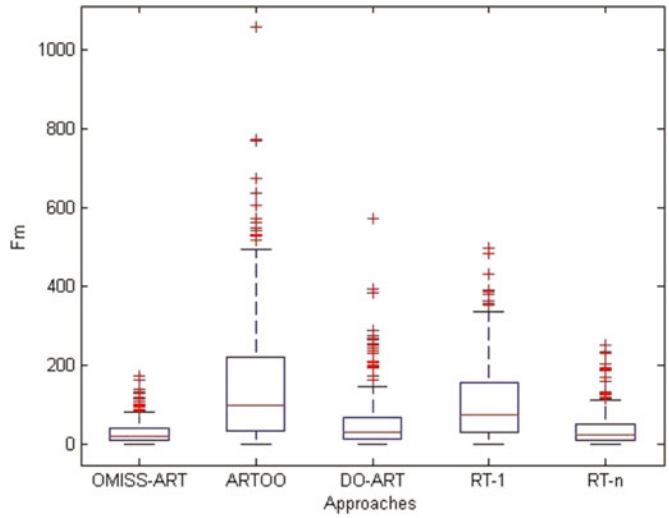

Fig. 20. Experimental result of F_m for Program #14.

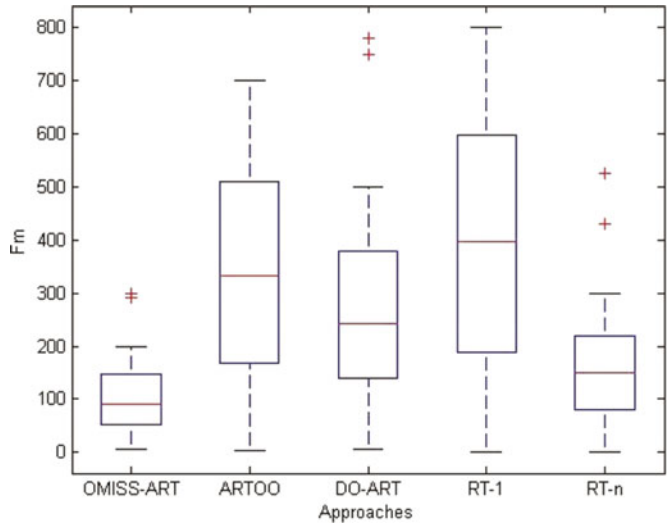

Fig. 21. Experimental result of F_m for Program #15.

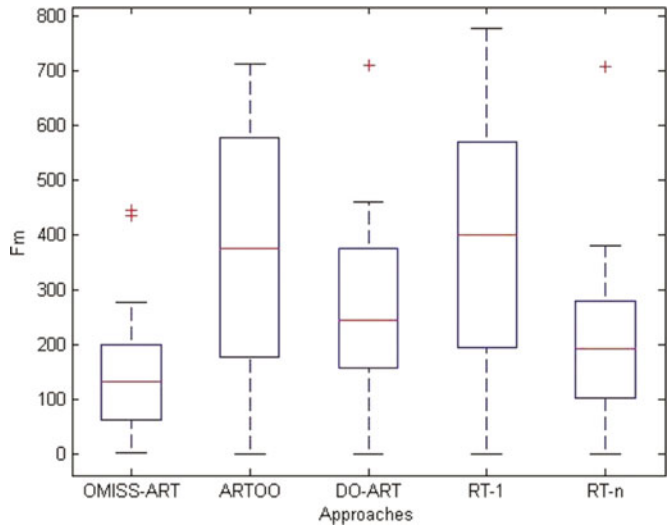

Fig. 22. Experimental result of F_m for Program #16.

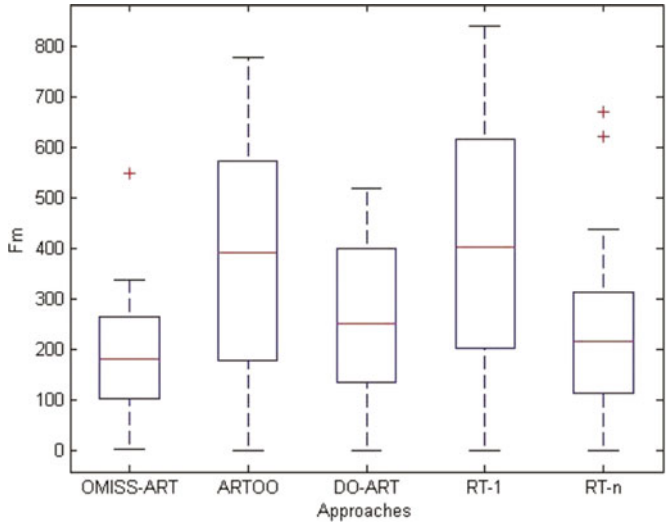

Fig. 23. Experimental result of F_m for Program #17.

RT-1 according to the mean F_m, based on all subject programs. Because OMISS differentiates test inputs for all the data types when evenly spreading test inputs, OMISS-ART consistently obtains better F_m results than RT-n.

Additionally, to investigate the impact of the total number of method invocations used (i.e., size of $MINV$) on F_m reported in Table VI, we calculated the total number of method calls used in each testing approach (see Table X). From Tables VI and X, we have the following observations.

1) Since every test input of OMISS-ART, DO-ART, and RT-n involved a sequence of method calls, but those of ARTOO and RT-1 involved only one, the total number of method calls for each of OMISS-ART, DO-ART, and RT-n were greater than that for ARTOO and RT-1. The main reason why OMISS-ART, DO-ART, and RT-n outperform ARTOO and RT-1 in terms of F_m is because those test inputs with more method calls generally improve the code coverage.
2) Although OMISS-ART employs 14% more method calls than ARTOO, it requires only 35% of the number of test cases to detect the first fault; OMISS-ART employs 5% more method calls than RT-1, but requires only 32% of the test cases to detect the first failure; DO-ART employs 71% more method calls than ARTOO, but only requires 57% of the number of test cases to detect the first fault; and DO-ART employs 57% more method calls than RT-1, but only requires 53% of the test cases to detect the first failure.
3) Although the total number of method calls for OMISS-ART is smaller than that for DO-ART and RT-n, OMISS-ART outperforms DO-ART and RT-n in terms of the F_m due to the even spreading of method invocations: it is shown that an even spreading of test inputs can improve the fault-detection effectiveness, especially with a good distance metric, such as is used in OMISS-ART.
4) Similar results and observations can be found in ARTOO and RT-1: ARTOO outperforms RT-1 because of its even spread of test cases.

TABLE IX
F_M AND F_M-TIME RATIOS FOR OMISS-ART

	F_m ratios				F_m-time ratios			
	OMISS-ART / ARTOO	OMISS-ART / DO-ART	OMISS-ART / RT-1	OMISS-ART / RT-n	OMISS-ART / ARTOO	OMISS-ART / DO-ART	OMISS-ART / RT-1	OMISS-ART / RT-n
Min **SUT ID**	0.21 (#14)	0.39 (#15)	0.20 (#12)	0.66 (#15)	2.12 (#4)	0.97 (#4)	2.18 (#4)	1.02 (#10)
Max **SUT ID**	0.54 (#1)	1.27 (#3)	0.46 (#10)	0.98 (#1)	6.22 (#14)	4.61 (#11)	8.08 (#1)	2.46 (#1)
Ave	0.35	0.71	0.33	0.83	3.78	2.72	4.51	1.62

TABLE X
TOTAL NUMBER OF METHOD CALLS, OVER 300 RUNS

SUT ID	OMISS ART	ARTOO	DO-ART	RT-1	RT-n
1	87290	38661	92751	58200	82704
2	2983	3032	4021	3140	3507
3	13937	12816	13842	13282	14228
4	4341	4218	4390	4407	4699
5	70258	52078	72112	59308	75407
6	31601	45747	35744	36587	36167
7	32552	27225	36363	37026	36801
8	25738	20542	27265	25919	26183
9	8208	6679	9726	7416	7911
10	2226	1612	16454	1558	2248
11	44729	27571	69025	24754	52351
12	78186	75920	84391	82229	80405
13	51052	36911	62219	46286	51165
14	32648	45139	50666	31978	37563
15	69685	102520	197142	119436	135368
16	128044	110988	210447	115946	1851744
17	146026	113274	254727	123161	216617
Sum	**829504**	**724933**	**1241285**	**790633**	**2715068**

TABLE XI
AVERAGE NUMBER OF DISTINCT FAULTS DETECTED (E_m)

SUT ID	E_m				
	OMISS-ART	ARTOO	DO-ART	RT-1	RT-n
#1	3.93	3.33	3.45	3.32	3.77
#2	5.00	4.00	5.00	4.00	5.00
#3	5.00	4.87	5.00	4.00	5.00
#4	4.00	4.00	4.00	3.00	4.00
#5	4.00	2.45	3.73	2.39	3.71
#6	4.00	3.00	3.71	2.50	3.81
#7	7.00	5.47	6.27	4.30	6.31
#8	8.93	5.45	8.69	5.23	8.88
#9	14.99	12.09	13.62	12.01	14.57
#10	13.00	7.86	8.47	7.35	12.61
#11	12.91	9.85	9.17	9.24	12.46
#12	21.00	15.27	18.24	15.50	20.36
#13	8.00	6.47	7.14	5.94	8.00
#14	10.56	8.38	8.75	8.36	9.21
#15	21.56	10.97	14.77	9.67	19.08
#16	22.19	12.99	17.45	11.51	20.38
#17	23.19	13.84	17.71	12.46	21.73
mean	11.13	7.66	9.13	7.10	10.52
sDev	6.98	4.24	5.50	4.21	6.46
error (95% CL)	± 0.21	± 0.13	± 0.17	± 0.13	± 0.20
uppB	11.34	7.79	9.29	7.23	10.72
lowB	10.92	7.54	8.96	6.98	10.33
Interval where 95% data fall	0.42	0.26	0.33	0.25	0.39
Accuracy%	± 1.90	± 1.67	± 1.82	± 1.79	± 1.86
max	28.00	21.00	25.00	21.00	26.00
min	3.00	0.00	1.00	0.00	1.00
range	25.00	21.00	24.00	21.00	25.00

2) Experiment II to Measure E_m: Table XI shows the average number of distinct faults (E_m) detected by 5000 test inputs for each testing approach. Again, this table is based on the data collected over 300 runs of tests for each subject program using different seeds. The statistical data (from "mean" to "range") is based on 5100 (300×17) datasets. In the experiment, the confidence level was set to 95%, giving the sample means, standard deviation (*sDev*), accuracy, and confidence intervals (*lowB* and *uppB*), as shown in Table XI.

From Table XI, we have the following observations.

1) OMISS-ART never found fewer faults than the other approaches for the same number of test inputs.
2) RT-n has the next best performance, followed by DO-ART, ARTOO, and RT-1.
3) The confidence intervals between the lower and upper bounds (*lowB* and *uppB*) do not overlap, for any testing approach, which implies that the E_m for OMISS-ART is greater than those of other approaches, with high probability.
4) The E_m was accurate to within $\pm 2\%$ of the mean value, with 95% confidence. Based on these observations, we can conclude that the E_m experimental results statistically have good reliability [46].

In order to further analyze the E_m of each testing approach for each subject program, Table XII summarizes the main statistical measures ($mean$, $sDev$, and $fPer$) for the 17 subject programs—where $fPer$ shows the percentage of faults detected with 5000 test inputs for each subject program. Figs. 24–40 show the experimental results in box plots for each subject program. From the statistical data, we have the following observations.

1) OMISS-ART never performs worse than other approaches in terms of the mean values of E_m.
2) In most cases, the standard deviation for OMISS-ART is the smallest (the exceptions being for Programs #16 and #17).
3) In most cases (except for Programs #14, #15, #16, and #17), the interquartile range (the box length) of OMISS-ART is equal to zero (which is reflected in the box plots having the top, bottom, and middle all coinciding).

This shows that OMISS-ART has very stable E_m values. For the four larger programs (Programs #14, #15, #16, and #17), the maximum and minimum (the two ends of the whiskers) are

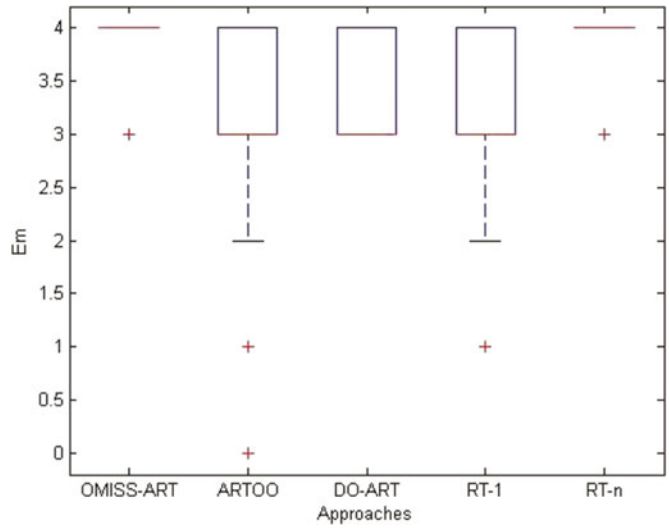

Fig. 24. Experimental result of E_m for Program #1.

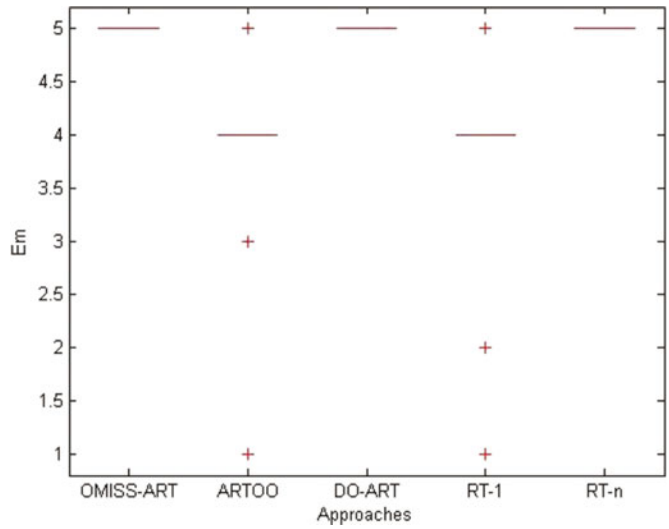

Fig. 25. Experimental result of E_m for Program #2.

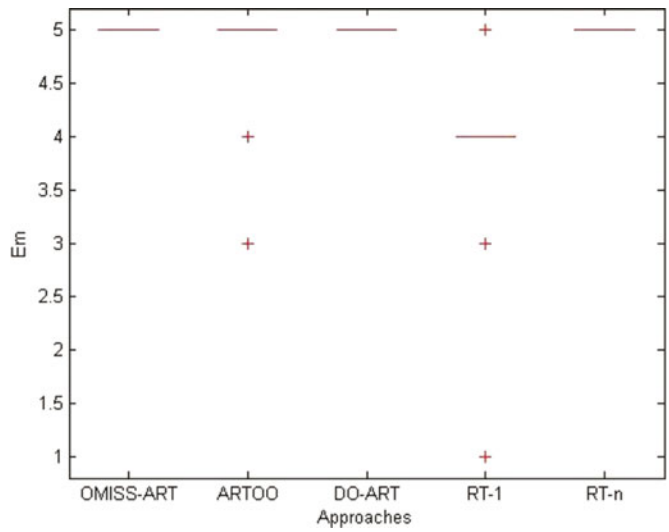

Fig. 26. Experimental result of E_m for Program #3.

TABLE XII
STATISTICAL RESULTS OF E_M FOR 17 SUBJECT PROGRAMS

SUT ID		OMISS -ART	ARTOO	DO -ART	RT-1	RT-n
#1	Mean	3.93	3.33	3.45	3.32	3.77
	sDev	0.26	0.65	0.50	0.62	0.42
	fPer(%)	98.25	83.17	86.25	82.92	94.17
#2	Mean	5.00	4.00	5.00	4.00	5.00
	sDev	0.00	0.68	0.00	0.69	0.00
	fPer(%)	100.00	80.00	100.00	80.00	100.00
#3	Mean	5.00	4.87	5.00	4.00	5.00
	sDev	0.00	0.36	0.00	0.71	0.00
	fPer(%)	100.00	97.40	100.00	80.00	100.00
#4	Mean	4.00	4.00	4.00	3.00	4.00
	sDev	0.00	0.00	0.00	0.63	0.00
	fPer(%)	100.00	100.00	100.00	75.00	100.00
#5	Mean	4.00	2.45	3.73	2.39	3.71
	sDev	0.00	0.68	0.67	0.69	0.72
	fPer(%)	100.00	61.33	93.25	59.75	92.75
#6	Mean	4.00	3.00	3.71	2.50	3.81
	sDev	0.00	0.31	0.69	0.60	0.55
	fPer(%)	100.00	75.00	92.75	62.42	95.33
#7	Mean	7.00	5.47	6.27	4.30	6.31
	sDev	0.00	0.92	0.58	1.20	0.46
	fPer(%)	100.00	78.14	89.57	61.38	90.10
#8	Mean	8.93	5.45	8.69	5.23	8.88
	sDev	0.25	0.96	0.71	1.16	0.34
	fPer(%)	99.26	60.52	96.52	58.11	98.63
#9	Mean	14.99	12.09	13.62	12.01	14.57
	sDev	0.10	1.44	0.50	1.29	0.52
	fPer(%)	99.93	80.58	90.82	80.07	97.11
#10	Mean	13.00	7.86	8.47	7.35	12.61
	sDev	0.00	1.06	0.92	1.42	0.57
	fPer(%)	100.00	60.49	65.18	56.54	96.97
#11	Mean	12.91	9.85	9.17	9.24	12.46
	sDev	0.36	1.01	1.24	1.20	0.70
	fPer(%)	92.21	70.38	65.51	65.98	89.00
#12	Mean	21.00	15.27	18.24	15.50	20.36
	sDev	0.00	3.01	2.75	2.85	1.24
	fPer(%)	100.00	72.73	86.87	73.81	96.95
#13	Mean	8.00	6.47	7.14	5.94	8.00
	sDev	0.00	0.90	0.57	1.09	0.00
	fPer(%)	100.00	80.88	89.21	74.29	100.00
#14	Mean	10.56	8.38	8.75	8.36	9.21
	sDev	0.85	0.86	0.89	0.89	0.85
	fPer(%)	88.03	69.83	72.89	69.69	76.78
#15	Mean	21.56	10.97	14.77	9.67	19.08
	sDev	1.79	2.05	4.41	2.28	2.52
	fPer(%)	89.83	45.72	61.54	40.31	79.50
#16	Mean	22.19	12.99	17.45	11.51	20.38
	sDev	2.54	1.98	4.25	2.32	2.19
	fPer(%)	85.33	49.96	67.13	44.28	78.40
#17	Mean	23.19	13.84	17.71	12.46	21.73
	sDev	3.19	2.49	4.62	2.89	2.87
	fPer(%)	82.81	49.44	63.24	44.51	77.60

always greater than those of other approaches, which shows that OMISS-ART performs better than other testing approaches.

Additionally, to investigate the impact of the number of method invocations on the E_m results reported in Table XI, we calculated the number of method calls in 5000 test inputs for each testing approach (see Table XIII), from which we have the following observations.

1) Since test inputs for OMISS-ART, DO-ART, and RT-n involved a sequence of method calls, but those for ARTOO and RT-1 involved only one, the total number of method calls for OMISS-ART, DO-ART, and RT-n were greater than that for ARTOO and RT-1.

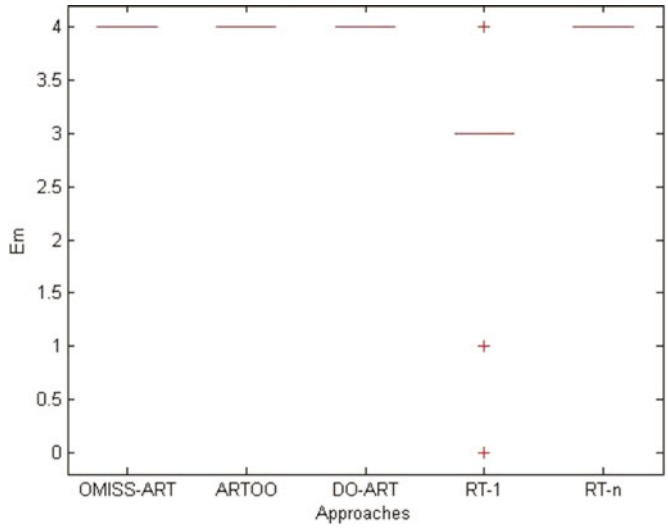

Fig. 27. Experimental result of E_m for Program #4.

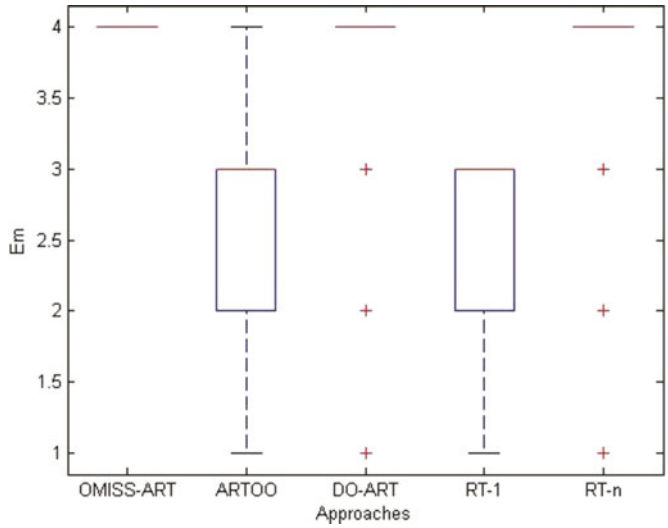

Fig. 28. Experimental result of E_m for Program #5.

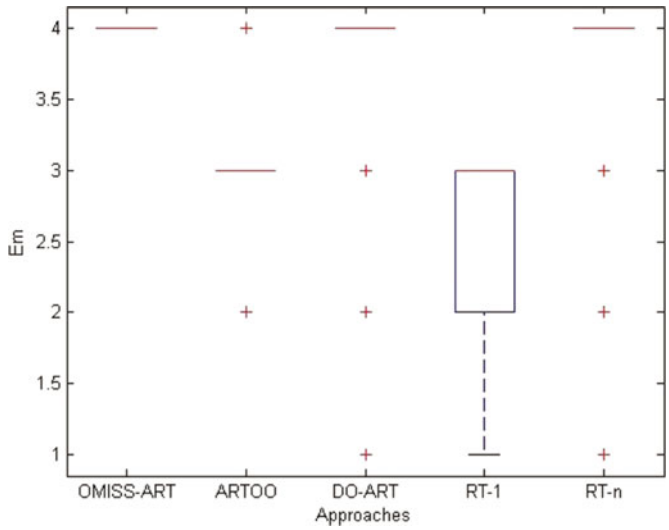

Fig. 29. Experimental result of E_m for Program #6.

TABLE XIII
TOTAL NUMBER OF METHOD CALLS (*MINV* SIZE) FOR EACH TESTING APPROACH, OVER 300 RUNS

SUT ID	OMISS- ART	ARTOO	DO- ART	RT-1	RT-n
1	4 266 001	1 500 000	4 650 928	1 500 000	5 247 390
2	4 617 064	1 500 000	4 391 472	1 500 000	5 232 427
3	3134968	1 500 000	4 533 284	1 500 000	2 870 650
4	3 050 748	1 500 000	4 498 792	1 500 000	3 178 748
5	4 747 366	1 500 000	4 598 439	1 500 000	5 341 942
6	3 228 475	1 500 000	4 509 030	1 500 000	3 673 603
7	5 307 716	1 500 000	4 505 780	1 500 000	5 350 326
8	5 186 694	1 500 000	4 976 929	1 500 000	5 342 729
9	5 201 134	1 500 000	4 529 852	1 500 000	5 439 405
10	4 359 535	1 500 000	4 504 068	1 500 000	4 500 210
11	5 162 748	1 500 000	4 802 293	1 500 000	5 248 734
12	5 262 663	1 500 000	4 750 511	1 500 000	5 248 790
13	5 534 135	1 500 000	4 864 961	1 500 000	5 249 939
14	4 510 429	1 500 000	4 393 281	1 500 000	4 336 979
15	5 136 332	1 500 000	4 950 184	1 500 000	5 362 359
16	4 964 538	1 500 000	5 125 687	1 500 000	5 235 772
17	5 345 691	1 500 000	5 235 176	1 500 000	5 398 264
Sum	79 016 237	25 500 000	79 820 667	25 500 000	82 258 267

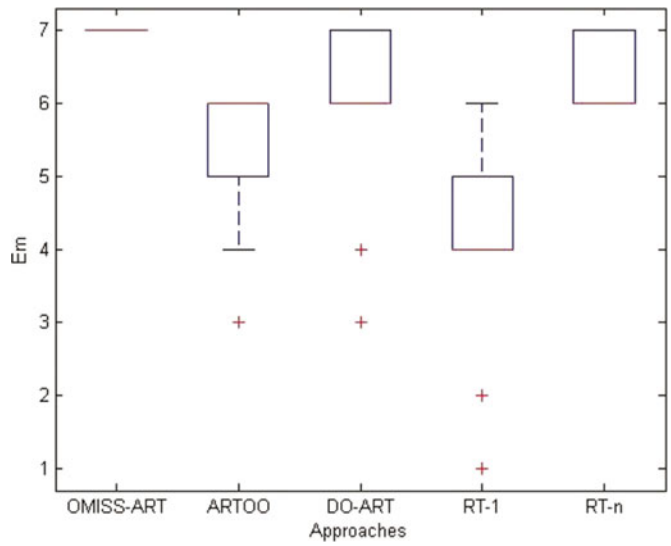

Fig. 30. Experimental result of E_m for Program #7.

2) For OMISS-ART, DO-ART, and RT-n, although the total number of method calls for OMISS-ART is less than that for DO-ART or RT-n, OMISS-ART outperforms DO-ART and RT-n because OMISS evenly spreads the method calls, but DO-ART and RT-n do not. It is shown that even spreading of test inputs can improve the fault-detection effectiveness with a better distance metric in OMISS-ART.
3) ARTOO and RT-1 have the same number of method calls because they only involve one method in each test input, but ARTOO outperforms RT-1. Thus, it is shown that an even spreading of objects, using the distance metric in ARTOO, can improve the fault-detection effectiveness.

Fig. 41 shows the average number of faults revealed by a number (n) of test inputs generated by each testing approach, among all subject programs. We found that E_m increases as n increases, and OMISS-ART outperforms all other approaches,

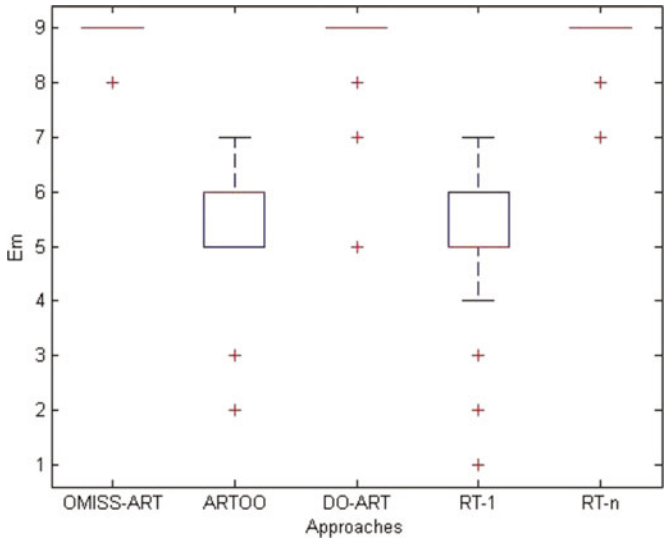

Fig. 31. Experimental result of E_m for Program #8.

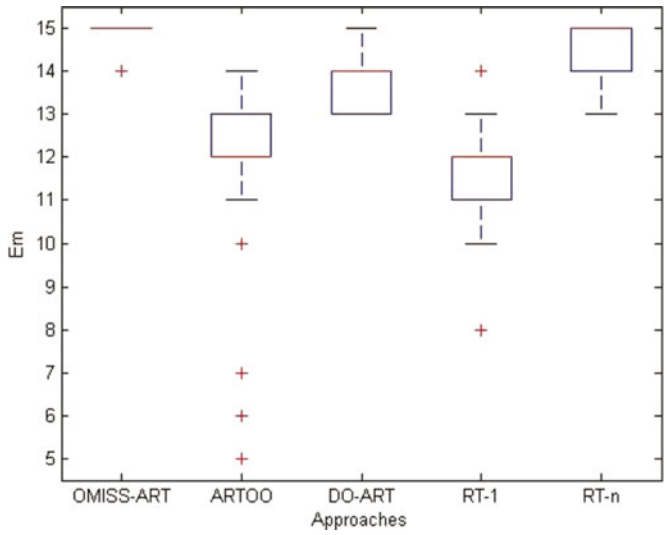

Fig. 32. Experimental result of E_m for Program #9.

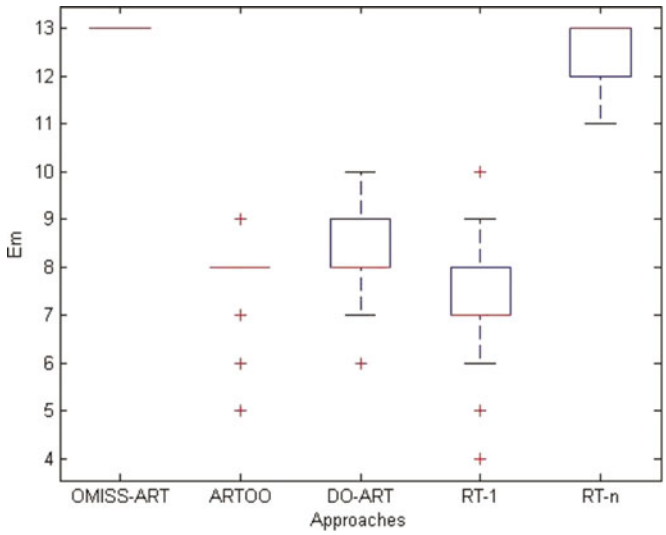

Fig. 33. Experimental result of E_m for Program #10.

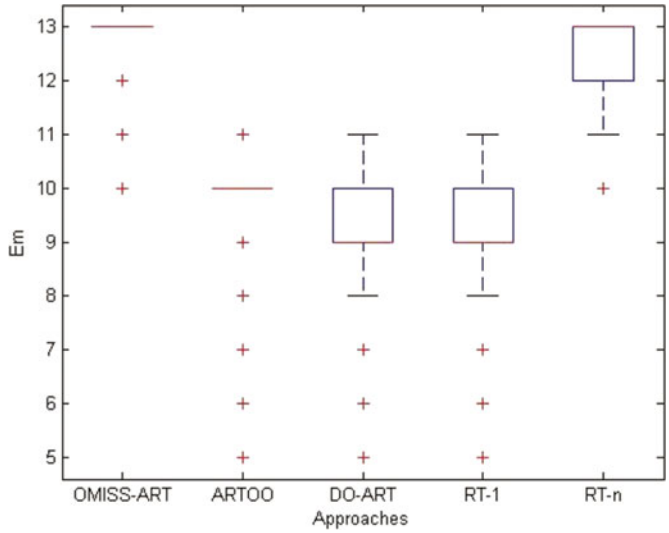

Fig. 34. Experimental result of E_m for Program #11.

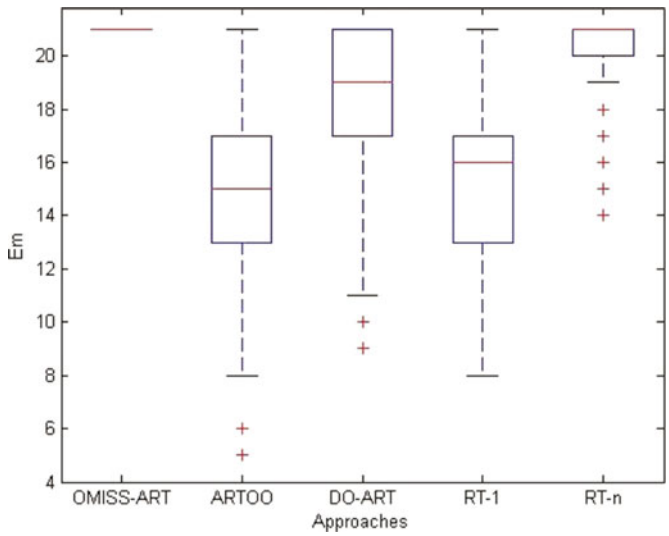

Fig. 35. Experimental result of E_m for Program #12.

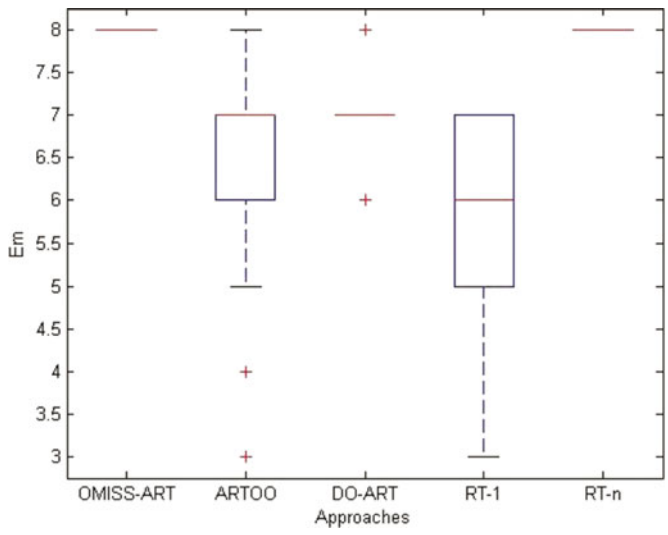

Fig. 36. Experimental result of E_m for Program #13.

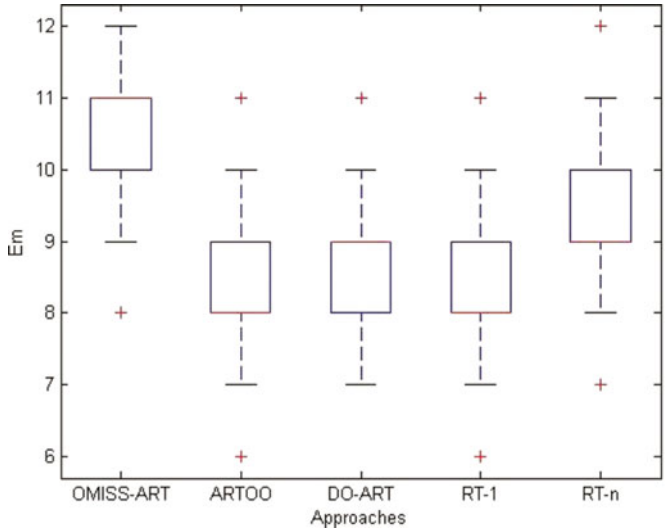

Fig. 37. Experimental result of E_m for Program #14.

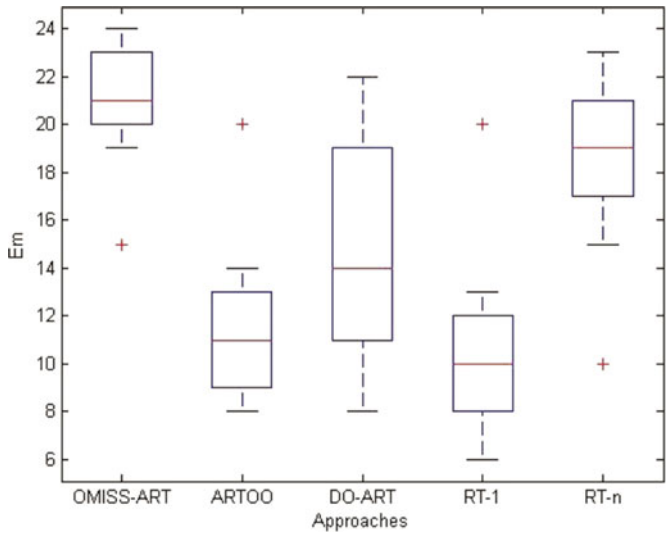

Fig. 38. Experimental result of E_m for Program #15.

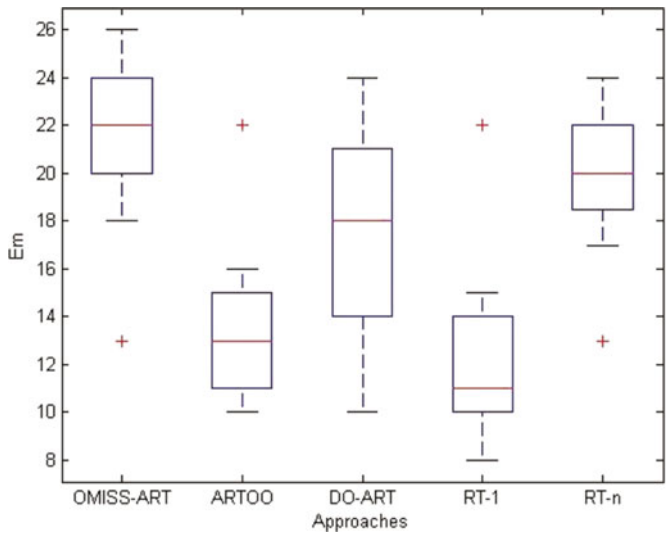

Fig. 39. Experimental result of E_m for Program #16.

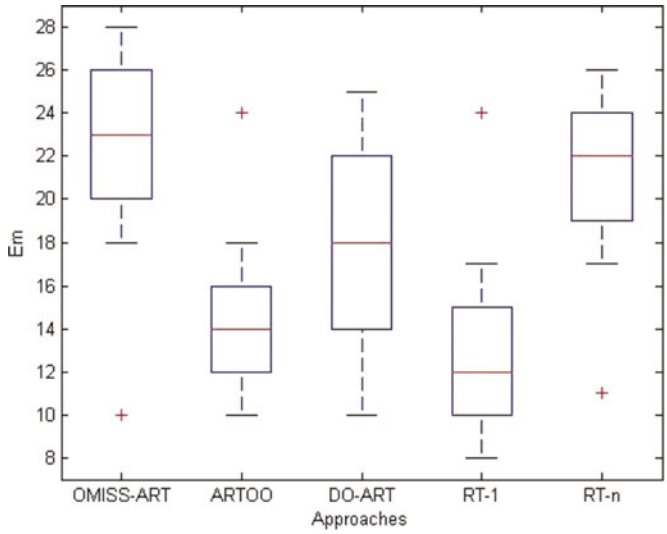

Fig. 40. Experimental result of E_m for Program #17.

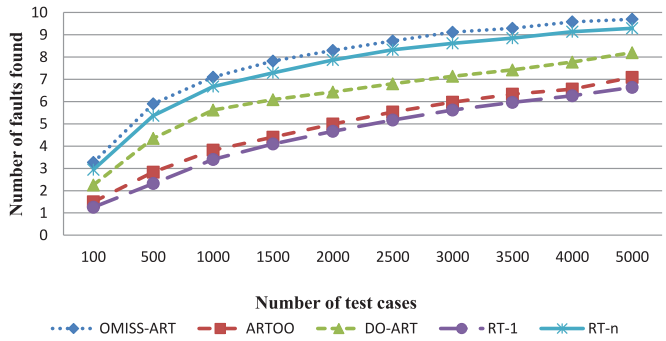

Fig. 41. Relationship between average number of faults found and number of test cases used for all 17 subject programs.

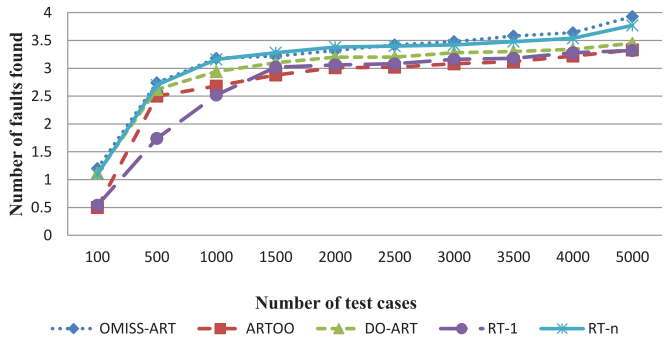

Fig. 42. Relationship between average number of faults found and number of test cases used for Program #1.

followed by RT-n, DO-ART, ARTOO, and RT-1, regardless of the value of n.

In order to further analyze the difference between different approaches for each program as the number of test cases increases, Figs. 42–58 are used to show the relationship between the average number of faults found and the number of test cases used for each subject program. Based on Table XII and Figs. 42–58, we have the following observations.

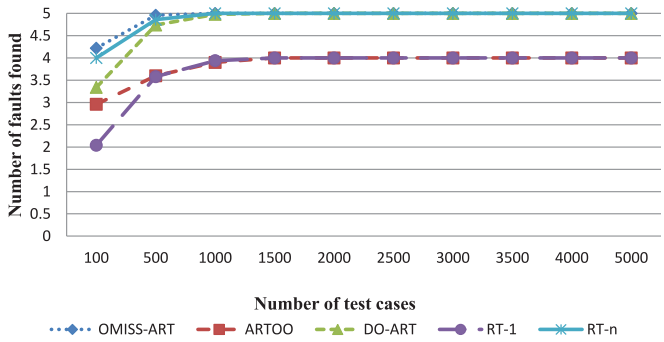

Fig. 43. Relationship between average number of faults found and number of test cases used for Program #2.

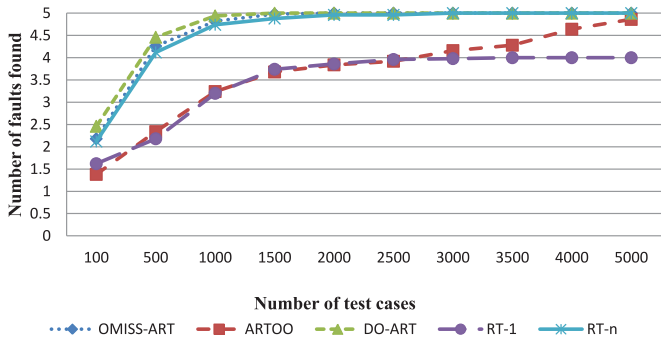

Fig. 44. Relationship between average number of faults found and number of test cases used for Program #3.

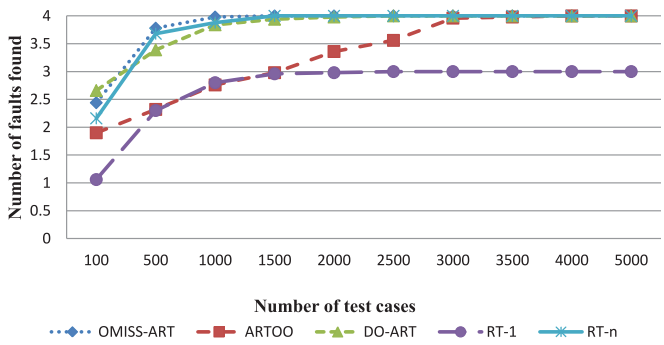

Fig. 45. Relationship between average number of faults found and number of test cases used for Program #4.

1) In most cases, OMISS-ART can find more faults than the other approaches for the same number of test cases.
2) Using up to 5000 test cases, OMISS-ART can detect all faults for Programs #2 to #7, #10, #12, and #13, and more than 80% of the faults for the other programs.
3) RT-n (also using up to 5000 test cases) can detect all faults for Programs #2–#4, and more than 70% of the faults for the other programs.
4) DO-ART (also using up to 5000 test cases) can detect all faults for Programs #2–#4, and more than 60% of the faults for the other programs.

The main reason why all the testing approaches could not detect all seeded faults after 5000 tests for some subject programs is that there were some interaction faults whose failure

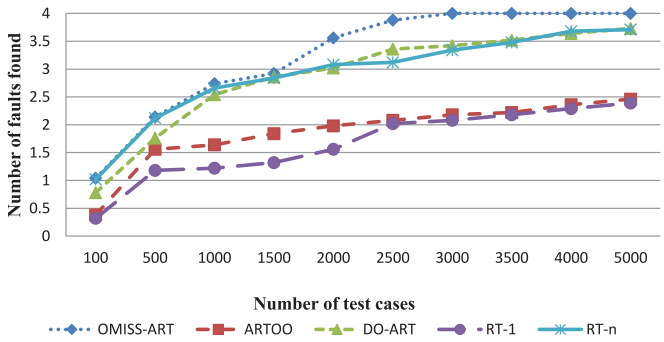

Fig. 46. Relationship between average number of faults found and number of test cases used for Program #5.

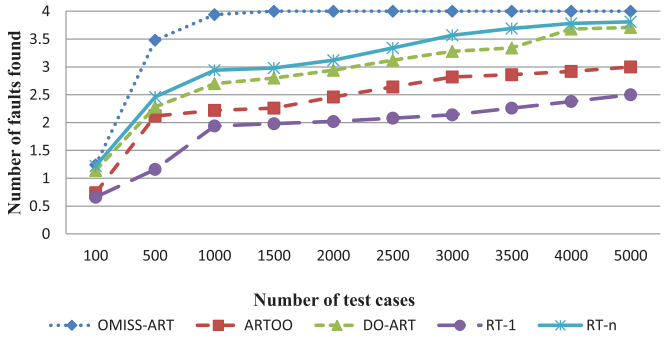

Fig. 47. Relationship between average number of faults found and number of test cases used for Program #6.

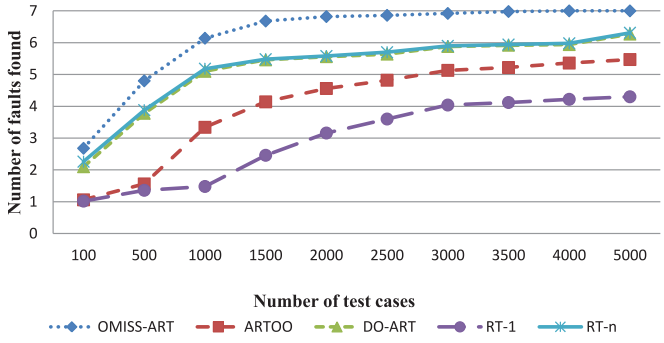

Fig. 48. Relationship between average number of faults found and number of test cases used for Program #7.

rates were very small. These interaction faults could only be triggered when special method sequences were invoked.

3) Discussion: In our empirical studies, we measured the failure-detection effectiveness and efficiency of each approach using the three metrics F_m, F_m-$time$, and E_m.

Our studies show that in most of cases, OMISS-ART requires the least number of test inputs to detect the first failure, and when using the same number of test inputs, it finds more faults than the other approaches. Although OMISS-ART outperforms other approaches in terms of finding faults, it also required more time (due to the distance calculations involved in determining the best test candidate). However, in most cases, the time taken to find the first failure by OMISS-ART is not more than twice that taken by RT-n.

OOS generally consists of multiple objects and multiple methods, and uses the object- and method-interaction to complete

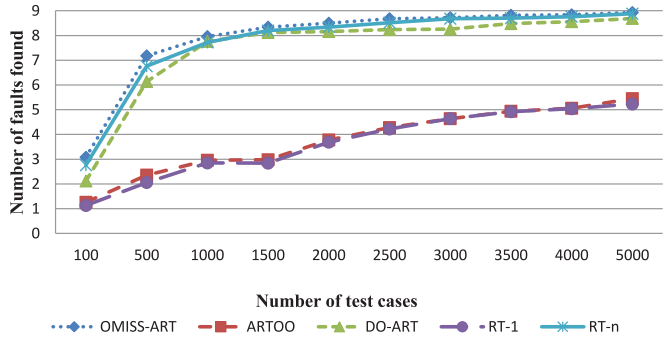

Fig. 49. Relationship between average number of faults found and number of test cases used for Program #8.

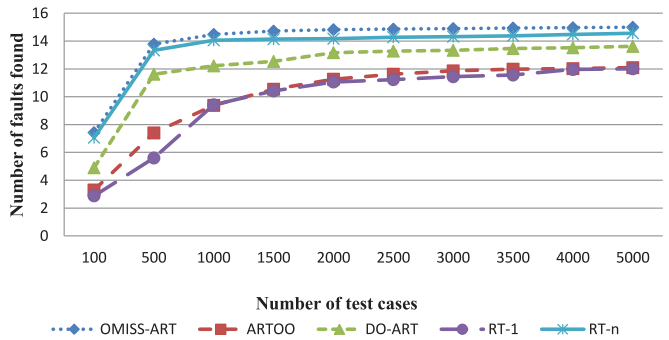

Fig. 50. Relationship between average number of faults found and number of test cases used for Program #9.

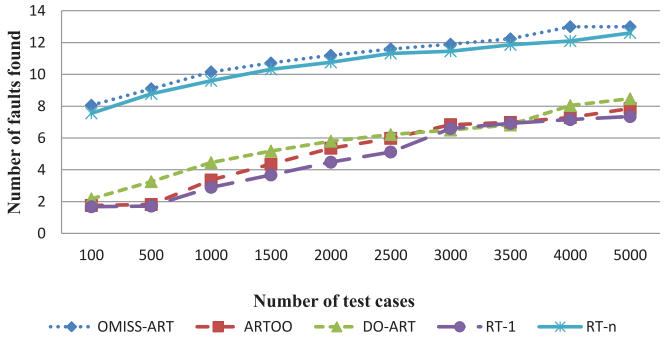

Fig. 51. Relationship between average number of faults found and number of test cases used for Program #10.

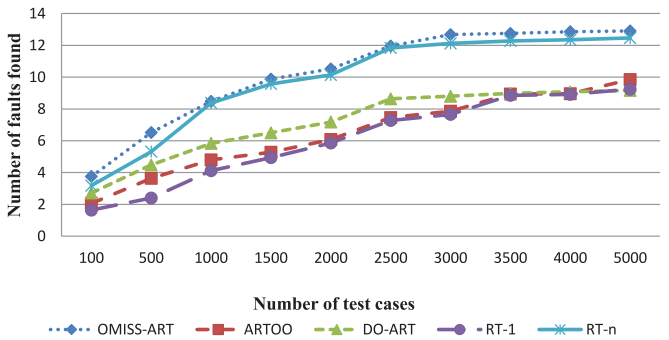

Fig. 52. Relationship between average number of faults found and number of test cases used for Program #11.

tasks. Any faults in these interactions can cause software failures, which are specific to OOS and only occur at runtime. These faults are difficult to detect but test cases consisting of method invocation sequences are more likely to reveal them [43], [44]. When comparing with ARTOO and RT-1 approaches which do not involve method invocation sequences, OMISS-ART, DO-ART, and RT-n were found to detect more faults, possibly because they revealed runtime interaction faults. Furthermore, using a more powerful distance metric, OMISS-ART was able to find faults faster, and find more faults with the same number of test cases, than both DO-ART and RT-n.

In this study, we used semantic information (class information and input structure) to construct and evenly spread the test cases. We did not refer to the runtime information when calculating the distance among arrays, pointers, and other dynamic attributes, and hence the calculations may have lost some degree of accuracy. Furthermore, our distance metric has the following two limitations.

1) Our *TCmSeqDist* metric differentiates between two sets of method sequences based on the following:
   a) their method names;
   b) the class of their methods' receivers; and
   c) the method signatures (i.e. types and numbers of parameters).
   This means that our approach treats $Bird1.fly(10000, Melbourne)$ and $Bird1.fly(100, Melbourne)$ as the same, which is not ideal.
2) The OMISS metric separates the distance calculation for object sets [see Formula (2)] from that for method invocation sequences [see Formula (3)], which means that Formula (3) cannot differentiate invocations of two methods based on their method receivers: Formula (3) treats $Bird1.fly(10000, Melbourne)$ and $Bird2.fly(10000, Melbourne)$ as the same, which is also not ideal.

Although our approach does not consider runtime information when measuring test input differences, it is, nonetheless, most effective at distinguishing objects of different classes and inputs involving method invocation sequences.

4) Threats to Validity: Despite our best efforts, our experiments may still face some threats to validity.

The first potential threat is the experimental setup. We have carefully designed the experiments such that OMISS-ART can be fairly compared with other existing testing approaches. After reimplementing ARTOO and divergence-oriented ART using .NET, we validated their correctness against the examples and results in the original papers [17], [18]. However, the conclusions from our empirical studies are based on 5100 datasets (300 runs for 17 subject programs), and since the datasets are larger than those in the original empirical studies [17], [18], the margins of error for the 95% confidence interval reported in Tables VI and XI are tighter.

The second potential threat relates to the selection of subject programs. The 17 programs we selected vary in size, and some may be considered smaller than typical industrial OOS. In spite of this, the subject programs have the following features: 1) the programs are typical OO programs constructed based on *classes* written in either C++ or C#, and 2) the program sizes

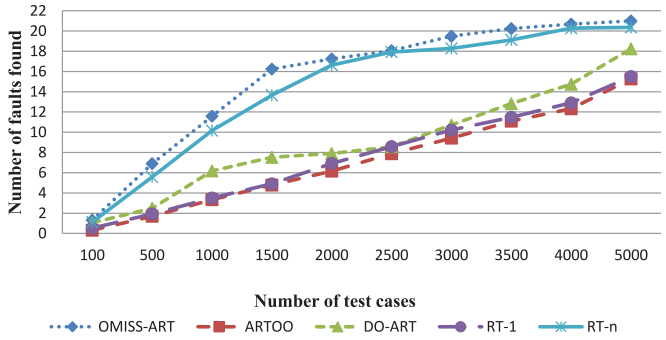

Fig. 53. Relationship between average number of faults found and number of test cases used for Program #12.

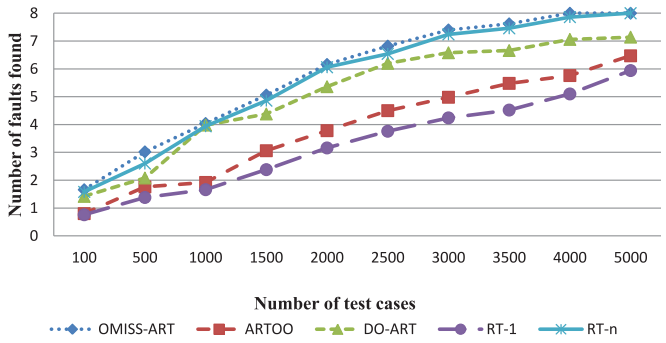

Fig. 54. Relationship between average number of faults found and number of test cases used for Program #13.

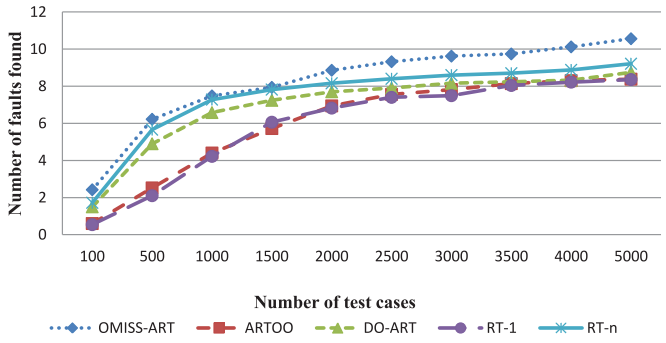

Fig. 55. Relationship between average number of faults found and number of test cases used for Program #14.

vary from 150 to 406 808 lines of code, allowing demonstration of OMISS-ART's scalability. Although, our OMISS-ART tool was implemented in C# and C++ based on the .Net framework, our OMISS metric, however, can be applied to all OO programs because it is based on OOS class and input structures, not on language-specific features.

A final potential threat may be that the mutants were generated by hand, which may be considered inappropriate. However, due to the current lack of good automatic mutation tools for C++ and C# programs, most researchers manually seed faults for subject programs written in these languages [42]. In this study, we applied a random number generator to select both the location and type of faults to be seeded, making the process semirandom and semiautomatic. During this process, some human effort was required to make some sensible judgments on how to insert appropriate types of faults in the specified location.

VI. Related Work

There are many OOS testing approaches, such as search-based testing, UML diagram-based testing, symbol execution-based testing, RT-based testing, and code coverage-based testing [1], [2]. Compared with other approaches, RT is very easy to use and requires less information to create test inputs. Hence, RT-based testing techniques have been widely applied to OOS testing. Unit testing of OOS based on RT techniques has drawn a lot of attention in the software testing community [5], [7]–[12], [22], [23]. Pacheco *et al.* [5], [7], [12] proposed a technique that improves RT by using information of previously executed test inputs to guide new test input generation. This information, which includes methods and related arguments as well as their execution traces and outputs, is used as feedback to avoid repeating entry into invalid states. This feedback-directed RT has been applied to some systems, and found to be more effective in finding faults than RT. Jaygarl *et al.* [47] improved feedback-directed RT by incorporating input-on-demand creation, coverage-based selection, and sequence-based-reduction techniques: they developed a tool, GenRed, which was able to improve branch coverage and effectively remove redundant test inputs. Both feedback-directed RT and GenRed are RT approaches that use runtime information to guide test case generation. In this study, we used the class information (static information) to construct a pool of random test inputs, from which we removed any inputs that triggered a runtime exception and crashed the software. Consequently, the final pool of test inputs which was used for all the testing approaches in our experiments contained only valid method invocation sequences.

ART [15], [19] has also been applied to testing OOS [16]–[18]. Ciupa *et al.* proposed a distance concept for objects, and a metric for measuring distances between objects [16], [17]. Their metric measures the distance between two objects of the same class, or two objects sharing the same ancestor, by considering the objects field distance, type distance, and recursive distance. They implemented their test approach, ARTOO, into a tool in the Eiffel language, and compared its performance with that of RT, finding ARTOO better than RT in terms of number of test inputs required to detect the first failure and the number of faults uncovered. Lin *et al.* [18] improved on ARTOO's code coverage by increasing the number of method invocations in each test input. They used the ARTOO distance metric to choose an object and generate a method receiver, using this method receiver to call more methods in that object. They implemented their approach, named divergence-oriented ART, as a tool, ARTGen, in the Java language, and found that it was able to find more faults with fewer test inputs than RT. However, the distance metric used in ARTOO and divergence-oriented ART can only be applied to a pair of objects. It cannot compare test inputs that involve multiple objects/methods of multiple classes and objects of different classes without common ancestor. Our approach uses a more comprehensive distance metric to support integration testing using ART.

d'Amorim *et al.* [48] integrated symbolic execution with operational models to form a model-based symbolic testing approach for OO unit testing. They conducted empirical studies

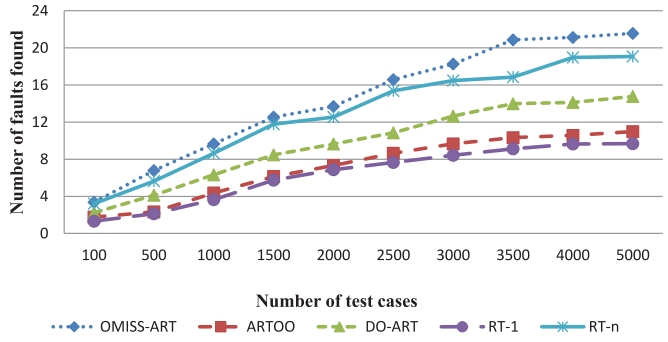

Fig. 56. Relationship between average number of faults found and number of test cases used for Program #15.

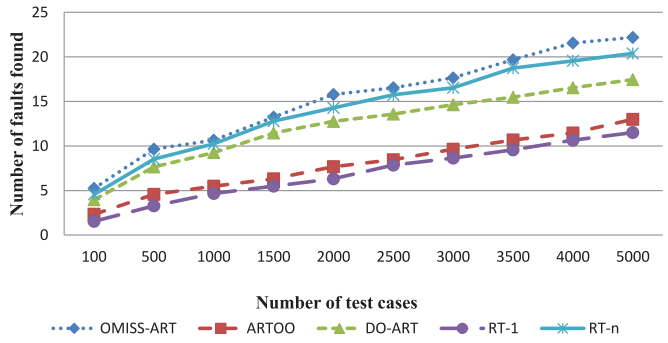

Fig. 57. Relationship between average number of faults found and number of test cases used for Program #16.

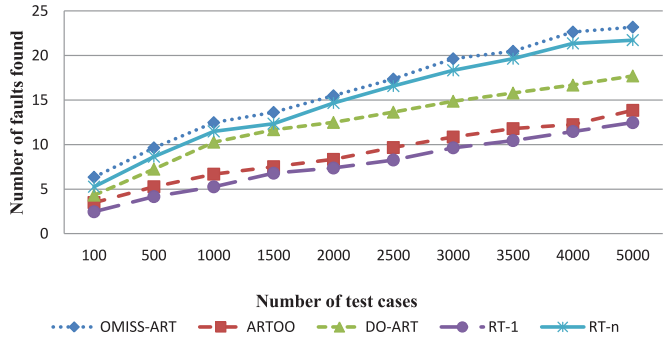

Fig. 58. Relationship between average number of faults found and number of test cases used for Program #17.

to compare the performance of their approach with that of three other techniques: model-based RT, exception-based symbolic testing, and exception-based RT. Their studies showed that RT performance depends on the number of test cases used, while symbolic execution performance depends on the power of the constraint solvers used. They recommended using all four techniques, arguing that they are complementary for revealing different faults. In contrast, our studies investigated an advanced RT strategy, ART, for testing OOS. Our approach employed a new distance metric to evenly spread test inputs, which is different from the approach proposed by d'Amorim *et al.* Our study shows that RT effectiveness can be improved by evenly spreading test cases.

There have also been studies into using RT to test interactive programs, which require that methods are run in a certain order, and hence satisfy the preconditions for running each method. Wei *et al.* [49] found that RT may leave some methods entirely untested due to its inability to satisfy the preconditions for these methods. To solve this, they proposed obtaining the preconditions for all methods by analyzing the source code: when testing a specific method r, they find a set (M) of methods that r depends on, and search for all related methods required to run all methods in M. At the end of this search, they construct a test sequence for all these methods found, and randomly select test inputs that can run this sequence. In our empirical studies, we were aware of the existence of preconditions for certain methods, and thus, we removed any unsafe or invalid method sequences (those causing exceptions) during pretesting, and used only those remaining ones in the test pool for the experiments.

The impact that the number of method invocations in the test cases has on the fault-detection effectiveness of RT, and the relationship between this and the consumption of testing resources when testing *interactive programs* have been examined [43], [44], with the conclusion that more invocations will consume more resources, but also improve fault detection. We also studied the impact of the number of method invocations on the effectiveness of RT. Test inputs involving a sequence of method calls have a higher chance of detecting the interaction faults than those involving a single method call. Additionally, our study shows that diversifying methods and their arguments can further improve the fault-detection effectiveness and efficiency.

VII. Conclusion and Future Work

Testing OOS is challenging because of the need to deal with new problems introduced by the special features of OO languages such as encapsulation, inheritance, and polymorphism. ART is an enhancement of RT, but requires a distance metric to differentiate candidate test cases. Ciupa *et al.* were the first to propose such a distance metric, and applied it to test individual methods in a single class; but their metric was not very well suited to evenly spreading test inputs that involved multiple objects/methods of multiple classes for integration testing.

In this paper, we have proposed a new distance metric OMISS that can handle test inputs involving multiple classes, objects, and methods, and can facilitate integration testing of OOS. We integrated this metric with the FSCS-ART algorithm to evenly spread test inputs to test mutants of 17 open source programs. Two series of empirical studies were conducted to compare our approach with ARTOO, divergence-oriented ART, and two variants of RT approaches, with results showing that our approach outperforms the others in terms of both the *F*-measure (the number of test inputs required to detect the first failure) and the *E*-measure (the total number of faults detected by a set of test inputs).

In the future, we will address the two limitations of OMISS metric discussed in Section V-B3. Additionally, we would like to consider runtime information when calculating distances, and enhance the functionality and efficiency of the OMISS-ART tool. We will also customize our OMISS-ART tool to support testing of OO software written in Java.

APPENDIX
DETAILED CALCULATION STEPS FOR *TestcaseDistance* BETWEEN T_1 AND T_3

Step	Formula_ID	Formula_Detail	Steps required for calculation (**SrC**) & Result of calculation (**RoC**)
1	1	$TestcaseDistance\,(T_1, T_3) = TCobjDist(T_1.OBJ, T_3.OBJ) + TCmSeqDist(T_1.MINV, T_3.MINV) = (2.8 + 0.22i) + 1.467 = 4.267 + 0.22i$	SrC = Steps 2 and 45 RoC = $4.267 + 0.22i$
2	2	$TCobjDist(T_1.OBJ, T_3.OBJ) = Min(ObjDist(T_1.OBJ, PL_1(T_3.OBJ)), ObjDist(T_1.OBJ, PL_2(T_3.OBJ)))$, where $PL_1(T_3.OBJ) = (Ps2, Dog1)$ and $PL_2(T_3.OBJ) = (Dog1, Ps2)$ $ObjDist(T_1.OBJ, PL_1(T_3.OBJ)) = ObjDist(Ps1, Ps2) + ObjDist(Bird1, Dog1) = 2.8 + 0.22i$ $ObjDist(T_1.OBJ, PL_2(T_3.OBJ)) = ObjDist(Ps1, Dog1) + ObjDist(Bird1, Ps2) = 10 + 0.5i$	SrC = Steps 3 and 24 RoC = $2.8 + 0.22i$
3	2	$ObjDist(T_1.OBJ, PL_1(T_3.OBJ)) = ObjDist(Ps1, Ps2) + ObjDist(Bird1, Dog1) = 0.1i + (2.8 + 0.12i) = 2.8 + 0.22i$	SrC = Steps 4 and 13 RoC = $2.8 + 0.22i$
4	4	$ObjDist(Ps1, Ps2) = BehDist(Ps1.\{null\}, Ps2.\{null\}) + AttDist(Ps1.\{Store1\}, Ps2.\{Store2\}) = 0 + 0.1i = 0.1i$	SrC = Steps 5 and 12 RoC = $0.1i$
5	6	$AttDist(Ps1.\{Store1\}, Ps2.\{Store2\}) = nonRefDist(Ps1.\{Store1\}, Ps2.\{Store2\}) + refDist(Ps1.null, Ps2.null) = 0.1i + 0$	SrC = Steps 6 and 11 RoC = $0.1i$
6	8	$nonRefDist(Ps1.\{Store1\}, Ps2.\{Store2\}) = typeDist(Ps1.\{null\}, Ps2.\{null\}) + secDist(Ps1.\{Store1\}, Ps2.\{Store2\})i = 0 + 0.1i = 0.1i$	SrC = Steps 7 and 10 RoC = $0.1i$
7	10	$secDist(Ps1.\{Store1\}, Ps2.\{Store2\})i = gSecDist(Ps1.\{Store1\}, Ps2.\{Store2\})i = 0.1i$	SrC = Step 8 RoC = $0.1i$
8	11	$gSecDist(Ps1.\{Store1\}, Ps2.\{Store2\})i = dist(Ps1.\{Store1\}, Ps2.\{Store2\})i = 0.1i$	SrC = Step 9 RoC = $0.1i$
9	12.5	$dist(Ps1.\{Store1\}, Ps2.\{Store2\}) = 1/range = 1/10 = 0.1$	RoC = 0.1
10	9	$typeDist(Ps1.\{null\}, Ps2.\{null\}) = 0$	RoC = 0
11	7	$refDist(Ps1.\{null\}, Ps2.\{null\}) = 0$	RoC = 0
12	5	$BehDist(Ps1.\{null\}, Ps2.\{null\}) = 0$	RoC = 0
13	4	$ObjDist(Bird1, Dog1) = BehDist(Bird1.\{fly(), expand(), grow()\}, Dog1.\{bark(), follow(), grow()\}) + AttDist(Bird1.\{Age = 3, Pet\_store = Ps1, Is\_sick = true\}, Dog1.\{Age = 10, Pet\_store = Ps2, Breed = good\}) = 0.8 + (2 + 0.12i) = 2.8 + 0.12i$	SrC = Steps 14 and 23 RoC = $2.8 + 0.12i$
14	6	$AttDist(Bird1.A, Dog1.A) = nonRefDist(Bird1.\{Age = 3, Is\_sick = true\}, Dog1.\{Age = 10, Breed = good\}) + refDist(Bird1.\{Pet\_store = Ps1\}, Dog1.\{Pet\_store = Ps2\}) = (2 + 0.07i) + (0.05i) = 2 + 0.12i$	SrC = Steps 15 and 22 RoC = $2 + 0.12i$
15	8	$nonRefDist(Bird1.\{Age = 3, Is\_sick = true\}, Dog1.\{Age = 10, Breed = good\}) = typeDist(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) + secDist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\})i = 2 + 0.07i$	SrC = Steps 16 and 19 RoC = $2 + 0.07i$
16	10	$secDist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\})i = gSecDist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\})i = \|3 - 10\|/100i = 0.07i$	SrC = Step 17 RoC = $0.07i$
17	11	$gSecDist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\})i = dist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\})i = 0.07i$	SrC = Step 18 RoC = $0.07i$
18	12.2	$dist(x, y) = \frac{\|x-y\|}{Range}$, $Range = 100$, $dist(Bird1.\{Age = 3\}, Dog1.\{Age = 10\}) = \|3 - 10\|/100 = 0.07$	RoC = 0.07
19	9	$typeDist(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) = TSizeDiff(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) + TSetDiff(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) = 0 + 2 = 2$	SrC = Steps 20 and 21 RoC = 2
20	9.1	$TSizeDiff(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) = \|1\text{-}1\| = 0$	RoC = 0
21	9.2	$TSetDiff(Bird1.\{Is\_sick = true\}, Dog1.\{Breed = good\}) = \|\{bool, string\}\| - \|null\| = 2 - 0 = 2$	RoC = 2
22	7	$refDist(Bird1.\{Pet\_store = Ps1\}, Dog1.\{Pet\_store = Ps2\}) = TCobjDist(Bird1.\{Pet\_store = Ps1\}, Dog1.\{Pet\_store = Ps2\}) * 1/2 = ObjDist(Ps1, Ps2) * 1/2 = 0.1i * 1/2 = 0.05i$	SrC = Step 4 RoC = $0.05i$
23	5	$BehDist(Bird1.M, Dog1.M) = \|size(Bird1.M) - size(Dog1.M)\| + (1 - size\{(Bird1.M) \cap (Dog1.M)\}/size\{(Bird1.M) \cup (Dog1.M)\}) = \|(\|\{fly(), expand(), grow()\}\| - \|\{bark(), follow(), grow()\}\|) + (1 - \|\{grow()\}\|/\|\{grow(), fly(), expand(), bark(), follow()\}\|) = \|3 - 3\| + (1 - 1/5) = 0 + 0.8 = 0.8$	RoC = 0.8
24	2	$ObjDist(T_1.OBJ, PL_2(T_3.OBJ)) = ObjDist(Ps1, Dog1) + ObjDist(Bird1, Ps2) = (4 + 0.5i) + (6 + 0i) = 10 + 0.5i$	SrC = Steps 25 and 36 RoC = $10 + 0.5i$
25	4	$ObjDist(Ps1, Dog1) = BehDist(Ps1.\{null\}, Dog1.\{bark(), follow(), grow()\}) + AttDist(Ps1.\{Store1\}, Dog1.\{Age = 10, Pet\_store = Ps2, Breed = good\}) = 1 + (3 + 0.5i) = 4 + 0.5i$	SrC = Steps 26 and 35 RoC = $4 + 0.5i$
26	6	$AttDist(Ps1.\{Store1\}, Dog1.\{Age = 10, Pet\_store = Ps2, Breed = good\}) = nonRefDist(Ps1.\{Store1\}, Dog1.\{Age = 10, Breed = good\}) + refDist(Ps1.\{null\}, Dog1.\{Pet\_store = Ps2\}) = (2 + 0.5i) + (1) = 3 + 0.5i$	SrC = Steps 27 and 34 RoC = $3 + 0.5i$
27	8	$nonRefDist(Ps1.\{Store1\}, Dog1.\{Age = 10, Breed = good\}) = typeDist(Ps1.\{null\}, Dog1.\{Age = 10\}) + secDist(Ps1.\{Store1\}, Dog1.\{Breed = good\})i = 2 + 0.5i$	SrC = Steps 28 and 31 RoC = $2 + 0.5i$

APPENDIX (CONTINUED)

28	10	$secDist(Ps1.\{Store1\}, Dog1.\{Breed = good\})i = gSecDist(Ps1.\{Store1\}, Dog1.\{Breed = good\})i = 5/10i = 0.5i$	SrC = Step 29 RoC = $0.5i$
29	11	$gSecDist(Ps1.\{Store1\}, Dog1.\{Breed = good\})i = dist(Ps1.\{Store1\}, Dog1.\{Breed = good\})i = 0.5i$	SrC = Step 30 RoC = $0.5i$
30	12.5	$dist(Ps1.\{Store1\}, Dog1.\{Breed = good\})$=5/$range$ = 5/ 10 = 0.5	RoC = 0.5
31	9	$typeDist(Ps1.\{null\}, Dog1.\{Age = 10\}) = TSizeDiff(Ps1.\{null\}, Dog1.\{Age = 10\}) + TSetDiff(Ps1.\{null\}, Dog1.\{Age$= 10$\})$ =1 + 1= 2	SrC = Steps 32 and 33 RoC = 2
32	9.1	$TSizeDiff(Ps1.\{null\}, Dog1.\{Age$= 10$\})$ = | 0-1| = 1	RoC = 1
33	9.2	$TSetDiff(Ps1.\{null\}, Dog1.\{Age = 10\}) = \|\|\{int\}\|-\|null\|\| = 1$	RoC = 1
34	7	$refDist(Ps1.\{null\}, Dog1.\{Pet\_store = Ps2\}) = TCobjDist(Ps1.\{null\}, Dog1.Pet\_store = Ps2) * 1/2 = ObjDist(null, Ps2)*1/2 = 2 * 1/2 = 1$	RoC = 1
35	5	$BehDist(Ps1.\{null\}, Dog1.\{bark(), follow(), grow()\}) = 1$	RoC = 1
36	4	$ObjDist(Bird1, Ps2) = BehDist(Bird1.\{fly(), expand(), grow()\}, Ps2.\{null\}) + AttDist(Bird1.\{Age = 3, Pet\_store = Ps1, Is\_sick = true\}, Ps2.\{Store2\}) = 1 + (5 + 0i) = 6 + 0i$	SrC = Steps 37 and 44 RoC = $6 + 0i$
37	6	$AttDist(Bird1.\{Age = 3, Pet\_store = Ps1, Is\_sick = true\}, Ps2.\{Store2\}) = nonRefDist(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) + refDist(Bird1.\{Pet\_store = Ps1\}, Ps2.\{null\}) = (4 + 0i) + (1) = 5 + 0i$	SrC = Step 38 and 43 RoC = $5 + 0i$
38	8	$nonRefDist(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) = typeDist(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) + secDist(Bird1.\{null\}, Ps2.\{null\})i = 4 + 0i$	SrC = Steps 39 and 40 RoC = $4 + 0i$
39	10	$secDist(Bird1.\{null\}, Ps2.\{null\})i = 0i$	RoC =$0i$
40	9	$typeDist(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) = TSizeDiff(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) + TSetDiff(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\}) = 1 + 3 = 4$	SrC = Steps 41 and 42 RoC = 4
41	9.1	$TSizeDiff(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\})$ = | 2-1|= 1	RoC = 1
42	9.2	$TSetDiff(Bird1.\{Age = 3, Is\_sick = true\}, Ps2.\{Store2\})$=$\|\{int, bool, string\}\|-\|null\|$ = | 3-0| = 3	RoC = 3
43	7	$refDist(Bird1.\{Pet\_store = Ps1\}, Ps2.\{null\}) = TCobjDist(Bird1.\{Pet\_store = Ps1\}, Ps2.\{null\}) * 1/2 = ObjDist(null, Ps1) * 1/2 = 2 * 1/2 = 1$	RoC = 1
44	5	$BehDist(Bird1.\{fly(), expand(), grow()\}, Ps2.\{null\}) = 1$	RoC = 1
45	3	$TCmSeqDist(T_1.MINV, T_3.MINV) = Size(T_1.\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly()\}, T_3.\{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\}) + MsD(T_1.\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly()\}, T_3.\{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\}) + SD(T_1.\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly()\}, T_3.\{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\}) = 0 + 0.8 + 0.667 = 1.467$	SrC = Steps 46, 47, and 48 RoC = 1.467
46	3.1	$Size(T_1.MINV, T_3.MINV) = \|length(\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly()\})\| - \|length(\{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\})\| = \|3 - 3\| = 0$	RoC = 0
47	3.2	$MsD(T_1.MINV, T_3.MINV) = (1 - \|\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly\} \cap \{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\}\|/\|\{Bird1.Grow(), Bird1.Expand(), Bird1.Fly\} \cup \{Dog1.Grow(), Dog1.Bark(), Dog1.Follow()\}\| = 1 - 1/5 = 0.8$	RoC = 0.8
48	3.3	$SD(T_1.MINV, T_3.MINV) = (0 + 1 + 1)/3 = 2/3 = 0.667$	RoC = 0.667

ACKNOWLEDGMENT

The authors would like to thank Y. Guo, X. Zhao, and S. Cai for their help in the experiments, and the anonymous reviewers for their comments on earlier versions of this paper.

REFERENCES

[1] R. V. Binder, "Testing object-oriented software: A survey," *Softw. Testing, Verification Rel.*, vol. 6, no. 3/4, pp. 125–252, Sep. 1996.

[2] M. Pezze and M. Young, "Testing object-oriented software," in *Proc. 26th Int. Conf. Softw. Eng.*, 2004, pp. 739–740.

[3] 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 Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 250–295, Jul. 1998.

[4] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013.

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 75–84.

[6] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Zhou, "A revisit of three studies related to random testing," *Sci. China Inf. Sci.*, vol. 58, no. 5, pp. 052104:1–052104:9, May 2015.

[7] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .Net with feedback-directed random testing," in *Proc. 2008 Int. Symp. Softw. Testing Anal.*, 2008, pp. 87–96.

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proc. 2007 Int. Symp. Softw. Testing Anal.*, 2007, pp. 84–94.

[9] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "An empirical study about the effectiveness of debugging when random test cases are used," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 452–462.

[10] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with MUT-aware sequence recommendation," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 293–296.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6, Jun. 2005, pp. 213–223.

[12] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. Companion 22nd ACM SIGPLAN Conf. Object-Oriented Program. Syst. Appl. Companion*, 2007, pp. 815–816.

[13] T. Y. Chen, F.-C. Kuo, and H. Liu, "Adaptive random testing based on distribution metrics," *J. Syst. Softw.*, vol. 82, no. 9, pp. 1419–1433, Sep. 2009.

[14] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code coverage of adaptive random testing," *IEEE Trans. Rel.*, vol. 62, no. 1, pp. 226–237, Mar. 2013.

[15] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Jan. 2010.

[16] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Object distance and its application to adaptive random testing of object-oriented programs," in *Proc. 1st Int. Workshop Random Testing*, 2006, pp. 55–63.

[17] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 71–80.
[18] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of Java programs," in *Proc. 2009 IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 221–232.
[19] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. 9th Asian Comput. Sci. Conf.*, 2004, pp. 320–329.
[20] M. M. Hassan and J. H. Andrews, "Comparing multi-point stride coverage and dataflow coverage," in *Proc. 2013 Int. Conf. Softw. Eng.*, 2013, pp. 172–181.
[21] D. C. Kung, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao, "Object-oriented software testing-some research and development," in *Proc. 3rd IEEE Int. High-Assurance Syst. Eng. Symp.*, 1998, pp. 158–165.
[22] M. Oriol and S. Tassis, "Testing .Net code with yeti," in *Proc. 2010 15th IEEE Int. Conf. Eng. Complex Comput. Syst.*, 2010, pp. 264–265.
[23] H. Y. Chen and T. H. Tse, "Equality to equals and unequals: A revisit of the equivalence and nonequivalence criteria in class-level testing of object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1549–1563, Nov. 2013.
[24] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 16, no. 04, pp. 553–584, Aug. 2006.
[25] T. Y. Chen, R. G. Merkel, G. Eddy, and P. K. Wong, "Adaptive random testing through dynamic partitioning," in *Proc. 4th Int. Conf. Qual. Softw.*, 2004, pp. 79–86.
[26] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Inf. Softw. Technol.*, vol. 46, no. 15, pp. 1001–1010, Dec. 2004.
[27] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing through test profiles," *Softw., Pract. Experience*, vol. 41, no. 10, pp. 1131–1154, Sep. 2011.
[28] J. Mayer, "Lattice-based adaptive random testing," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.,* 2005, pp. 333–336.
[29] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal voronoi tessellations—A new approach to random testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 163–183, Feb. 2013.
[30] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Trans. Rel.*, vol. 58, no. 4, pp. 619–633, Dec. 2009.
[31] A. F. Tappenden and J. Miller, "Automated cookie collection testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, Feb. 2014, Art. no. 3.
[32] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Phys. Doklady*, vol. 10, no. 8, pp. 707–710, Feb. 1966.
[33] "Doxygen: Source code documentation generator tool," 2013. [Online]. Available: http://www.doxygen.org
[34] "Understand: Source code analysis & metrics," 2013. [Online]. Available: http://www.scitools.com/index.php
[35] "Microsoft visual studio," 2013. [Online]. Available: https://www.visualstudio.com
[36] "Codeforge-free open source codes forge and sharing," 2013. [Online]. Available: http://www.codeforge.com
[37] "Sourceforge-download, develop and publish free open source software," 2013. [Online]. Available: http://sourceforge.net
[38] "Codeplex-open source project hosting," 2013. [Online]. Available: http://www.codeplex.com
[39] "Codeproject-for those who code," 2013. [Online]: Available: http://www.codeproject.com
[40] "Github, where software is built," 2015. [Online]. Available: https://github.com
[41] K. P. Chan, T. Y. Chen, and D. Towey, "Forgetting test cases," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, 2006, vol. 1, pp. 485–494.
[42] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.
[43] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Proc. 2008 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 19–28.
[44] A. Arcuri, "Longer is better: On the role of test sequence length in software testing," in *Proc. 2010 3rd Int. Conf. Softw. Testing, Verification Validation*, 2010, pp. 469–478.
[45] A. Derezinska, "Quality assessment of mutation operators dedicated for C# programs," in *Proc. 2006 6th Int. Conf. Qual. Softw.*, 2006, pp. 227–234.
[46] Y. Liu and H. Zhu, "An experimental evaluation of the reliability of adaptive random testing methods," in *Proc. 2nd Int. Conf. Secure Syst. Integr. Rel. Improvement*, 2008, pp. 24–31.
[47] H. Jaygarl, K.-S. Lu, and C. K. Chang, "GenRed: A tool for generating and reducing object-oriented test cases," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf.*, 2010, pp. 127–136.
[48] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 59–68.
[49] Y. Wei, S. Gebhardt, B. Meyer, and M. Oriol, "Satisfying test preconditions through guided object selection," in *Proc. 2010 3rd Int. Conf. Softw. Testing, Verification Validation*, 2010, pp. 303–312.

Jinfu Chen (M'13) received the B.E. degree from Nanchang Hangkong University, Nanchang, China, in 2004, and the Ph.D. degree from Huazhong University of Science and Technology, Wuhan, China, in 2009, both in computer science.

He is an Associate Professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software analysis, and trusted software.

Dr. Chen is a Member of the ACM and the China Computer Federation.

Fei-Ching Kuo (M'06) received the Bachelor's of Science (Hons.) degree in computer science and the Ph.D. degree in software engineering, both from Swinburne University of Technology, Hawthorn, VIC, Australia.

She was a Lecturer at the University of Wollongong, Wollongong, NSW, Australia. She is currently a Senior Lecturer in the Department of Computer Science and Software Engineering, Swinburne University of Technology. She was also the Program Committee Chair for the 10th International Conference on Quality Software 2010 and the Guest Editor of a special issue of the *Journal of Systems and Software*, special issue of the *Software Practice and Experience*, and special issue of the *International Journal of Software Engineering and Knowledge Engineering*. Her current research interests include software analysis, testing, and debugging.

Tsong Yueh Chen (M'03) received the B.Sc. and M.Phil. degrees from the University of Hong Kong, China, the M.Sc. and Diploma degrees from the Imperial College of Science and Technology, London, U.K., and the Ph.D. degree from the University of Melbourne, Australia.

He is currently a Professor of software engineering in the Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, VIC, Australia. He is the originator of metamorphic testing and adaptive random testing. His current research interests include software testing, debugging, and program repair.

Dave Towey (M'03) received the B.A. and M.A. degrees in computer science, linguistics, and languages from the Trinity College, University of Dublin, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from the University of Hong Kong, China.

He is currently an associate professor at the University of Nottingham Ningbo China, in Zhejiang, China, where he also serves as the director of teaching and learning for the School of Computer Science. His current research interests include software testing and technology enhanced teaching and learning. He cofounded the ICSE International Workshop on Metamorphic Testing in 2016.

Dr. Towey is a member of the ACM.

Chenfei Su received the B.E. degree in computer science in 2012 from Jiangsu University, Zhenjiang, China, where he is currently working toward the Master's degree in the School of Computer Science and Communication Engineering.

His research interests include software testing and service computing.

Rubing Huang (M'12) received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology, Wuhan, China, in 2013.

He is currently an Assistant Professor in the Department of Software Engineering, School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. He has more than 20 publications in journals and proceedings including the *Journal of Systems and Software*, *Information and Software Technology*, the *International Journal of Software Engineering and Knowledge Engineering*, *Security and Communication Networks*, the IEEE Computer Society International Conference on Computers, Software & Applications, the International Conference on Software Engineering and Knowledge Engineering (SEKE), ACM Symposium on Applied Computing, etc. His current research interests include software testing and software maintenance, especially combinatorial testing, random testing, adaptive random testing, and test case prioritization.

Dr. Huang has served as the Program Committee Member of SEKE2014, SEKE2015, and SEKE2016. He is a member of the ACM and the China Computer Federation.