MorphyCloud: A Service Agent for Automating Datamorphic Testing

Hong Zhu
School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, United Kingdom
email:hzhu@brookes.ac.uk

October 17, 2022

Abstract

Morphy is a test automation framework and environment based on datamorphic testing methodology of software testing, which can be applied to all kinds of software including AI and machine learning applications. This paper presents a service agent implemented in Java that re-wraps Morphy's core functions as services. This enables distributed testing on a cluster of computers through commands as well as test scripts written in any scripting language using Morphy's high level functions as the basic test instructions. The paper demonstrates by examples that it significantly improves the test efficiency via parallel execution of test scripts and the scalability of automated testing via distributing test tasks to a cluster of computers. Moreover, it also significantly enhances the power of test systems in the datamorphic testing methodology, since test scripts can be written in any expressive script languages, and meta test scripts can be written to control testing as a multi-agent distributed system.

Keywords: *Software test, Test automation, Artificial Intelligence, Test tools, Datamorphic testing, Morphy, Cloud computing, Services*

1 Introduction

Datamorphic testing has been proposed as an approach to automate software testing, especially testing AI applications [12, 13]. In this approach, test automation focuses on the development, evolution and operation of a test system to achieve testing purposes. Experiments have demonstrated that this approach is effective and efficient in testing AI applications through test automation at three different levels of abstraction, i.e. at test activities, strategies and processes levels [1, 3, 12, 13].

Morphy is a test automation framework and environment that are designed and implemented based on the datamorphic testing methodology. Through Morphy's graphic user interface, a tester can load a test system written in Java, manage test entities, and perform testing on an AI application by invoking test morphisms, applying predefined test strategies that are implemented as Morphy's tools, and recording and replaying test scripts to achieve test automation. The experiments also demonstrated that the approach enables test code to be reused, composed and constructed to reduce the cost of testing [7–10].

In our experiments with datamorphic testing method and the Morphy testing tool in real industry case studies, it is recognised that it is desirable to execute testing in the service-oriented approach because the test sets are often so large that testing will benefit from an approach that adopts the big data and cloud computing techniques. This document presents *MorphyCloud*, which is a service agent version of Morphy's test runner with a command line interface. It employs service-oriented distributed cloud computing technique in testing so that distributed test automation can be easily achieved via writing and executing test scripts in any scripting languages.

The paper is organised as follows. Section 2 is an overview of datamorphic testing methodology, the Morphy architecture and core functions, as well as its framework for the construction of test systems.

2 Preliminary

This section defines the basic concepts of the datamorphic testing method, presents Morphy's architecture, functionality and its framework for constructing test systems in Java.

2.1 Datamorphic Testing

Datamorphic testing is a systems engineering approach to software testing in which the notion of *test system* plays the central role. Software testing processes are regarded as a process in which a test system is developed, evolved and operated to achieve the goal of software testing [7, 8, 10]. A test system is defined as an algebraic structure of the software artefacts involved in testing. These artefacts are classified into two kinds: *entities* and *morphisms*.

Test entities are objects and data that are created, processed, stored, and used in testing. Typical examples of test entities include *test cases* (TC), *test suites* (TS), the *program under test* (P), and *test reports* (TR), etc.

Test morphisms are mappings between test entities. They create, process and/or transform test entities to achieve testing objectives. They can be implemented by writing test code, invoked to perform test activities and composed to automate testing strategies and processes. Typical examples of test morphisms are test generators, test metrics, test oracles, and test result analysers, etc.

Formally, a *test system* $\mathscr{T} = \langle \mathscr{E}, \mathscr{M} \rangle$ consists of a set $\mathscr{E}$ of test entities and a set $\mathscr{M}$ of test morphisms on $\mathscr{E}$.

In particular, the following categories of test morphisms have been recognised.

1. Test Generators

   A *test generator* generates test cases. Here, we distinguish the following four types of test generators according to their inputs.

   (a) A *seed maker* g generates a test suite without input of existing test cases, but uses information contained in other types of test entities, such as from program code and/or formal specification, or even without any input such as random test case generators and fuzzers [5]. So, it is a mapping onto test suites $g : E \to TS$ from some test entity E or even void.

(b) A *datamorphism* d transforms existing test cases into new test cases. Therefore, for $K \geq 1$, a K-ary *datamorphism* $d : TC^K \rightarrow TC$ is a mapping from K test cases $t_1, \cdots, t_K \in TC$ to one test case $t \in TC$. Datamorphisms are called *data mutation operators* in our previous work [4].

(c) A *test set filter* φ is a mapping from test sets to test sets, i.e. $\varphi : TS \rightarrow TS$. For example, it could be used to remove redundant test cases in a test set T for regression testing. A test set filter can often be defined via a predicate on test cases, which is called a *test case filter* defined below.

(d) A *test case filter* φ is a mapping from test cases and test sets to truth values, such as a predicate that determines whether a test case t should be included in a test set T, i.e. $\varphi : TS \times TC \rightarrow \{True, False\}$.

2. *Test Oracles*

A test oracle determines if the program under test behaves correctly on test cases. There are a few common forms of test oracles.

(a) *Test Cases with Expected Outputs*. The most common form of test oracles are test cases that each test case t contains an input and its expected output. The output of the program on test input is then compared with the expected output. If they match, the program P is correct, otherwise the program fails on the test cases. Formally, a test oracle $p(t)$ can be defined as a predicate on test cases t with expected output $t.output$ as follows.

$$p(t) \Leftrightarrow P(t.input) = t.output$$

where $t.input$ is the input of the test case t, and $P(t.input)$ is the output of program P on the input.

For testing machine learning applications, labelled data can be used as such test cases.

(b) *Predicates on Test Case*. Another simple form of test oracle is a predicate $p(x)$ on test cases x without expected output, i.e. a mapping $p : TC \rightarrow \{True, false\}$ from test cases to truth values. For example, given a specification of program P with a pair of pre-condition $pre(x)$ on input value x and post-condition $post(x, y)$ on input x and output y, i.e. $pre(x)$ $\{P\}$ $post(x, y)$, the corresponding test oracle can be represented as a predicate

$$pre(x) \Rightarrow post(x, P(x)),$$

where x is the input, and $P(x)$ is the output of program P on input x.

(c) *Metamorphic Relations*. A test oracle can be defined as an axiom that asserts a relation on a number K of the inputs $x_1, \cdots, x_K$ and their corresponding outputs of the program under test [2]. They can be represented as a K-ary predicate p on test cases, i.e. a mapping $p : TC^K \rightarrow \{True, false\}$ from K test cases to truth values, where $K > 0$. Such a relation is called a *metamorphic relation*. For example, an axiom in an algebraic specification in the form of a conditional equation

$$cond(x_1, \cdots, x_k) \Rightarrow exp_L(x_1, \cdots, x_k) = exp_R(x_1, \cdots, x_k)$$

is in fact a metamorphic relation. For instance, consider the following algebraic law of $sin(x)$ function.

$$(y = \pi - x) \Rightarrow (sin(x) = sin(y))$$

The following is a binary metamorphic relation R for testing a program that implements the function $sin(x)$.

$$\langle x, y\rangle \in R \Leftrightarrow (y = \pi - x) \wedge (sin(x) = sin(y)) \tag{1}$$

(d) *Metamorphisms.* Given a K-ary datamorphism d, and a $K+1$-ary metamorphic relation p, we can define an unary predicate p_d such that for all test cases x, $p(x)$ if and only if there are K test cases $x_1, \cdots, x_k$ such that $x = d(x_1, \cdots, x_K)$ and $p(x_1, \cdots, x_K, x)$. In other words,

$$p_d(x) \Leftrightarrow x = d(x_1, \cdots, x_K) \wedge p(x_1, \cdots, x_K, x).$$

This specific form of test oracles derived from datamorphisms and metamorphic relation are called *mutational metamorphic relations* in our previous work [6] and *metamorphisms* in datamorphic testing method [13]. For example, the metamorphic relation given in Equ (1) can be represented as a metamorphism p for program $sin(x)$ as follows,

$$p(x) \Leftrightarrow sin(x) = sin(d(x))$$

where $d(x) = \pi - x$.

There are a number of advantages of metamorphisms over general format metamorphic relations. First, we can apply the datamorphisms on existing test cases to generate a mutant test case and then to check the correctness of the program on the mutant test cases. Thus, testing can be more efficient. Second, as demonstrated by Morphy, a test automation framework can be easily implemented to support metamorphisms then to support the more general metamorphic relations without loss too much generality. Finally, but more importantly, metamorphic relations can be relatively easily derived from datamorphisms.

3. *Test Metrics*

A *test metric* is a measurement of test cases or test suites. There are two types of test metrics:

(a) A *test case metric* μ is a mapping from test cases to real numbers, often in the context of a test set, i.e. $\mu : TS \times TC \rightarrow R$. For example, measuring the similarity of a test case t to others in a test set T is such a test metrics. It can be used to reduce the sizes of test suite.

(b) A *test set metric* Ψ is a mapping from test sets to real numbers, i.e. $\Psi : TS \rightarrow R$. For example, the code coverage and adequacy measurements of test sets are such test set metrics [11].

4. *Test Executers*

A *Test Executer* ε executes the program P under test on a test case and receives the output from the program. Mathematically speaking, let $Input$ and $Output$ be the input domain and output of the program under test. It is a mapping $\varepsilon : (Input \rightarrow Output) \times Input \rightarrow Output$ from input data to output as defined by the program, i.e. $\varepsilon_P(x) = P(x)$. In other words, it is a functor in category theory.

5. *Test Result Analysers*

A *Test Result Analyser* α analyses test results and generates test reports. Thus, it is a mapping $\alpha : TS \rightarrow TR$ from test sets to test reports. Typical examples of test result analysers are:

(a) statistical evaluators that calculate the failure rate of the program on test cases,
(b) bug reports that list the bugs detected by testing, and
(c) visualisations of program execution results that display test results graphically.

Datamorphic testing regards software testing as an engineering process in which a test system is developed, evolved and applied to achieve the purposes of testing. A software test system can be specified by defining the set of test entities involved and the set of morphisms used according to the testing methods and techniques to be employed. Morphy supports datamorphic testing by providing a framework for testers to implement test systems in programming language Java. It is given in the next subsection.

2.2 Morphy's Framework for Defining Test Systems

Morphy's framework of test systems consists of a set of abstract and concrete classes so that test code can be integrated with them to form a complete test environment. These classes implement the key concepts of datamorphic testing as Java classes, although the program under test does not need be written in Java [9].

The generic class **TestCase** given in code Listing (1) defines the notion of test cases with two type parameters for the input and output datatypes.

Listing 1: Skeleton of The Generic Class *TestCase*

```
public class TestCase<InputType,OutputType> {
  public String id; /* The id number */
  public InputType input; /*  The input data of the test case */
  public OutputType output; /* The output data of the test case */
  public TestDataFeature feature; /* Feature: "original" | "mutant" */
  public String type; /* The name of the method generated it*/
  public List<String> origins; /* TCs ids that it is generated from */
  public String correctness; /* Correctness: "MM_Name:(pass|fail);" */
  public TestCase() { ... } /* The constructor of a new test case */
  public String toString() { ... } /** Convert into a string */
  public TestDataFeature strToFeature(String featureStr) {...}
  public List<String> getOrigins() { .. }
  public void setOrigins(List<String> origins) {...}
  public String getType() { ... }
  public void setType(String type) { ...}
  public TestDataFeature getFeature() { ...}
  public void setFeature(TestDataFeature feature) { ... }
}
```

The **TestCase** class contains attributes for representing (a) the ID of the test case, (b) the input data, and (c) the output data. Moreover, it also contains an attribute called **feature** for how the test case is generated. It can be "original", i.e. a seed test case generated by a seed maker; or "mutant", which means it is generated by a datamorphism. The **type** attribute of the test case gives the datamorphism's name or the seed makers's name that generated it. When it is a mutant, the **origins** attribute gives the IDs of the test cases that the mutants is generated from. The **correctness** attribute of the test case records the outcomes of check the correctness of the test case against metamorphisms. The correctness is recorded in a format as follows:

$$\{metamorphismName : (pass|fail), \}^*$$

A set of abstract and concrete methods are also defined by the **TestCase** class for setting and getting the attributes as well as to convert the test cases between String and value representations.

The generic class **TestPool** with input and output datatypes as parameters defines the notion of test suite, which consists of a list of test cases. The code skeleton of the generic class is shown in Listing (2).

Listing 2: Skeleton of The Generic Class *TestPool*

```
public class TestPool<InputType,OutputType> {
  /** Container to hold a test set */
  public List<TestCase<InputType,OutputType>> testSet;

  /** Constructor to create a new empty set pool */
  public TestPool() { ... }

  /** Add a new seed test case with a value as the input.
   * @param v the value of the input
   */
  public String addInput(InputType v) { ... }

  /** Add a test case into the test pool
   * @param tc: the test case to be added
   */
  public void addTestCase(TestCase tc) { ... }

  /** Remove a test case from the test pool
   * @param tc: the test case
   */
  public void removeTestCase(TestCase tc) { ... }

  /** Remove one test case from the test pool
   * @param id: the identity of the test case;
   */
  public void removeTestCase(String id) { ... }

  /** Remove a set of test cases from the test pool
   * @param tcList: the test cases to be removed
   */
  public void removeAllTestCases(List<TestCase> tcList) { ... }

  /** Set the output value of the test case
   * @param v: the value of the output of the test case
   * @param tcIndex: the id number of the test case
   */
  public void setOutput(OutputType v, String id) { ... }

  /** Get the test case of a given id number
   * @param index: id number of the test case
   * @return: the test case of the id number in the test pool
   */
  public TestCase<InputType,OutputType> get(String id){ ... }

  /** Convert the test pool into a string */
  public String toString() { ... }
}
```

It defines an attribute to hold a list of test cases and provides methods for adding and removing test cases to/from the test suite.

In Morphy, a test system is realised as a Java class with test entities as attributes and test morphisms as methods. In such a class, an attribute of type **TestPool** can be annotated by metadata **@TestSetContainer** to indicate that it holds the test suite to perform testing. Additional attributes of type **TestPool** can be included as auxlary test sets, but they must not be annotated as **@TestSetContainer**. Each test morphism is annotated with metadata to indicate its kind as shown in Table 1.

Table 1: Annotations of Test Morphisms

Morphism	Annotation	Parameter	Return
Seed Maker	@SeedMaker	Nil	Void
Datamorphism	@Datamorphism	TestCase	TestCase
Metamorphism	@Metamorphism	TestCase	Boolean
Test Case Metrics	@TestCaseMetrics	TestCase	Real
Test Case Filter	@TestCaseFilter	TestCase	Boolean
Test Set Metrics	@TestSetMetrics	Nil	Real
Test Set Filter	@TestSetFilter	Nil	Nil
Test Executer	@TestExecuter	Input	Output
Analyser	@Analyser	Nil	Void

The Java classes that declare this annotation metadata can be found in the GitHub repository for this project; see Footnote[1] for the repositopry's URL.

The next subsection illustrates the above with a simple example.

2.3 Example of Test System in Morphy Format

A simple example of test system is given below for testing the trigonometric function $sin(x)$ in Java's **Math** library.

Listing 3: An Example of Test System in Morphy Format

```
package morphy.examples;
import morphy.annotations.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.JOptionPane;

public class SinTest {

  @TestSetContainer(
    inputTypeName = "Double",
    outputTypeName = "Double"
  )
  public TestPool<Double,Double> testSuite=new TestPool<Double,Double
      >();

  @TestExecuter
  public Double testSinX(Double x) {
    return Math.sin(x);
  }

```

[1]https://github.com/hongzhu6129/MorphyCloud.git

```
@MakeSeed
public void randomValues(){
  Random randomGenerator = new Random();
  for (int i=0; i<100;i++){
    TestCase<Double, Double> tc=new TestCase<Double,Double>();
    tc.input = randomGenerator.nextDouble()*Math.PI/2;
    tc.feature = TestDataFeature.original;
    tc.setType("randomValues");
    testSuite.addTestCase(tc);
  }
};

@Datamorphism
public TestCase<Double,Double> piMinus(TestCase<Double,Double> seed){
  TestCase<Double,Double> mutant = new TestCase<Double,Double>();
  mutant.input = Math.PI - seed.input;
  return mutant;
}

@Datamorphism(filter = "tooClose")
public TestCase<Double,Double> mid(TestCase<Double,Double> x1,
    TestCase<Double,Double> x2) {
  TestCase<Double,Double> mutant = new TestCase<Double,Double>();
  mutant.input = (x1.input+x2.input)/2;
  return mutant;
}

@TestSetMetric
public double avgDistance() {
  double total =0;
  double num = 0;
  for (TestCase<Double,Double> x:testSuite.testSet){
    total += x.input;
    num++;
  };
  if (num==0) {return 0;}
  else {return total/num;}
}

@TestSetFilter
public void sparse() {
  List<TestCase> toBeRemovedTCs = new ArrayList<TestCase>();
  int testSetSize = testSuite.testSet.size();
  for (int i=0; i<testSetSize -1; i++) {
    TestCase<Double,Double> x=testSuite.testSet.get(i);
    Double distanceX = null;
    for (int j=i+1; j<testSetSize; j++) {
      TestCase<Double,Double> y=testSuite.testSet.get(j);
      double distanceXY = Math.abs(x.input - y.input);
      if (distanceX==null){distanceX = distanceXY;
      }else{
        if (distanceX>distanceXY){distanceX=distanceXY;}
      }
    };
    if (distanceX != null && distanceX <0.00001) {
      toBeRemovedTCs.add(x);
    }
    }
    testSuite.removeAllTestCases(toBeRemovedTCs);
  }

  @TestCaseMetric
  public double distance(TestCase<Double,Double> x) {
    Double distance = null;
    for (TestCase<Double,Double> y: testSuite.testSet) {
      if (x.id.equals(y.id)) {continue;}
      double distanceXY =  Math.abs(y.input - x.input);
      if (distance == null) {distance=distanceXY;
      }else{
        if (distanceXY<distance){distance=distanceXY;}
      }
    };
    return distance;
  }

  @TestCaseFilter
  public Boolean tooClose(TestCase<Double,Double> x) {
    return (distance(x) <0.00001);
  }

  @Metamorphism(
      applicableTestCase="mutant",
      applicableDatamorphism = "piMinus",
      message="Failed␣the␣rule:␣Sin(x)␣=␣Sin(pi␣-␣x)")
  public boolean PiMinusAssertion(TestCase<Double,Double> tc){
    TestCase<Double, Double> originalTc = testSuite.get(tc.getOrigins()
        .get(0));
    return (Math.abs(tc.output - originalTc.output) <=
        0.000000000000001);
  }

  @Analyser
  public void visualization() {
    new VisualizeOutput(testSuite.testSet);
  }
}
```

In this test system, the class **Double** of real numbers is used for both the input and output datatypes. The attribute **testSuite** of type **TestPool<Double, Double>** stores the test suite. A seed maker called **randomValue** generate 100 random numbers in the range from 0 to $\pi/2$ as input values for the test cases.

The test system contains two datamorphisms: **piMinus**, which generates a test case with input value $\pi - x$ from a test case whose input value is x, and **mid**, which generates a test case with input value $(x + y)/2$ from two test cases with input values x and y. It also has a metamorphism **piMinusAssertion** which asserts that the output value of a mutant test case y must be equal to the output value of its original test case x, from which y has been generated by applying the **piMinus** datamorphism. In other words, it asserts that $sin(x) = sin(\pi - x)$.

The test system also defines one test set metric method **avgDistance** to calculate the average distances between test cases, one test set filter method **sparse** that removes test cases closer than 10^{-5} from their nearest neighbour, one test case metric method **distance** that calculates the distance from a test case to its nearest neighbour, and one test case filter method **tooClose**, which returns **true** if that distance is less than 10^{-5}.

The method **testSinX** is the test executer, which invokes the Java **Math** library function **sin** on a **Double** real number input x and returns the result produced.

2.4 Morphy's Architecture and Functions

Morphy is a test automation environment that enables testers to automate software testing in the datamorphic approach. In this subsection, we give a brief introduction to the architecture and main functions of Morphy.

2.4.1 Architecture

Morphy's architecture as shown in Figure 1. In addition to the graphic user interface, it contains three main components: the *test manager*, the *test runner* and the *test scripting* components.

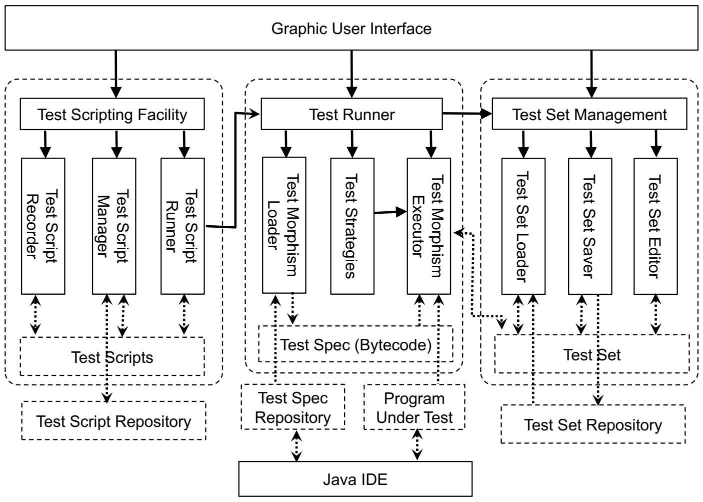


Figure 1: The Architecture of Morphy

The *test manager* enables test sets to be loaded from file, viewed, edited and saved to file.

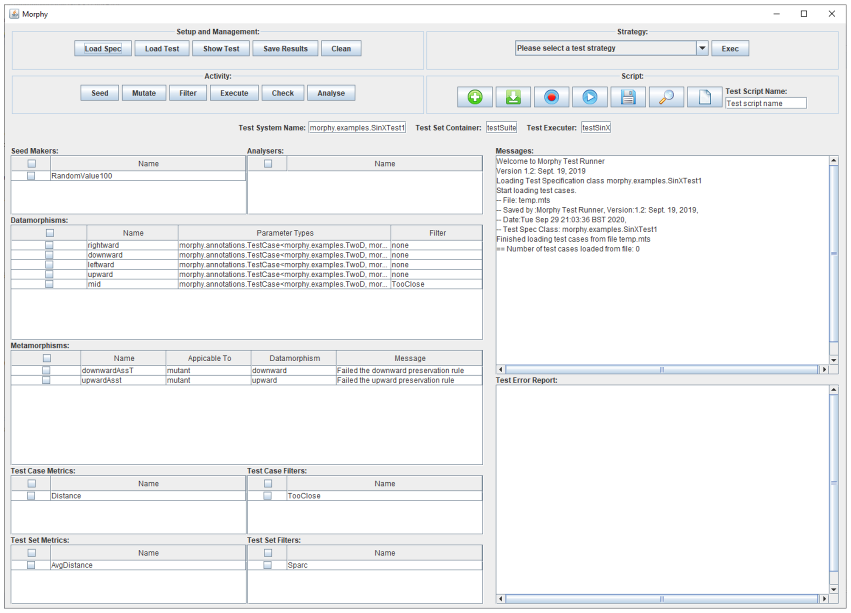
The *test runner* component enables a test system to be loaded onto Morphy so that test morphisms can be invoked to perform testing activities. It also implements various test strategies that combine test morphisms to achieve advanced test automation. The results of executing test morphisms, such as running the program under test on test cases and the checking of program correctness on test cases via invocations of metamorphisms, are recorded and saved into the test set. Test systems can be developed with any Java IDE, but a wizard has been developed as an Eclipse plug-in to generate a test system as a new skeleton Java class.

The *test scripting* component enables interactive testing activities to be recorded as test scripts, saved into files, reloaded from files, edited and replayed. It provides a facility for test automation at process level.

Morphy retains its state between executions so when the system is started, it restores the state from the last time it existed. When a new test system is loaded, it will initialise its state.

2.4.2 Graphic User Interface

Morphy provides a Graphic User Interface (GUI) to testers for managing testing assets and performing testing tasks. Its main window is shown in Figure 2. It forms a user-friendly environment in which testing artefacts can be managed, basic testing activities can be performed and automated testing facilities can be invoked.



Please select a test strategy

Test Script Name: Test script name

Test System Name: morphy.examples.SinXTest1 Test Set Container: testSuite Test Executer: testSinX

Seed Makers:

☐	Name
☐	RandomValue100

Analysers:

☐	Name

Datamorphisms:

☐	Name	Parameter Types	Filter
☐	rightward	morphy.annotations.TestCase<morphy.examples.TwoD, mor...	none
☐	downward	morphy.annotations.TestCase<morphy.examples.TwoD, mor...	none
☐	leftward	morphy.annotations.TestCase<morphy.examples.TwoD, mor...	none
☐	upward	morphy.annotations.TestCase<morphy.examples.TwoD, mor...	none
☐	mid	morphy.annotations.TestCase<morphy.examples.TwoD, mor...	TooClose

Metamorphisms:

☐	Name	Applicable To	Datamorphism	Message
☐	downwardAssT	mutant	downward	Failed the downward preservation rule
☐	upwardAsst	mutant	upward	Failed the upward preservation rule

Test Case Metrics:

☐	Name
☐	Distance

Test Case Filters:

☐	Name
☐	TooClose

Test Set Metrics:

☐	Name
☐	AvgDistance

Test Set Filters:

☐	Name
☐	Sparc

Messages:

```
Welcome to Morphy Test Runner
Version 1.2: Sept. 19, 2019
Loading Test Specification class morphy.examples.SinXTest1
Start loading test cases.
-- File: temp.mts
-- Saved by :Morphy Test Runner, Version:1.2: Sept. 19, 2019,
-- Date:Tue Sep 29 21:03:36 BST 2020,
-- Test Spec Class: morphy.examples.SinXTest1
Finished loading test cases from file temp.mts
== Number of test cases loaded from file: 0
```

Test Error Report:

Figure 2: Morphy's Main GUI

At the very top of Morphy's main window are four panels of buttons for setting up and managing the test system, performing test activities by invoking various types of test morphisms, applying test strategies, and recording/replaying test scripts.

The left-hand side below that is a collection of tables of test morphisms which can be selected with checkboxes to perform automated testing interactively. To the right of that is a message panel that reports the execution and outcomes of testing activities, and another that reports errors detected by checking test results against metamorphisms.

2.4.3 Main Functions

The Morphy provides the following functions to support test automation at the activity level. These functions can be invoked through the *Setup And Management* and *Activity* panels of the main window.

The *test management* functions are:

- *Load Spec*: Load a test system implemented as a Java class file.
- *Load Test*: Load a previously saved test set file and add its test cases to the current test suite.
- *Show Test*: Open a Test Suite window for viewing and/or manually editing test set. Test case metrics will be invoked when test cases are viewed.

- *Save Results*: Save the current test suite to a file.
- *Clean*: Re-initialise the system and remove all test cases from the test suite.

The *test activity* functions invoke various test morphisms, which can be interactive performed through the buttons in the Activity panel of the GUI.

- *Seed*: Invoke selected seed maker morphism(s) to generate a set of seed test cases and add them to the current test suite.
- *Mutate*: Apply selected datamorphism(s) to the current test suite to generate mutant test cases and add them to the current test suite.
- *Filter*: Apply selected test set filter(s) to modify the current test suite.
- *Measure*: Apply selected test set metrics to measure the test quality; the measurement result is shown as a pop-up.
- *Execute*: Run the test executer, which runs the program under test on the test cases in the current test suite. The results will be saved to the output field of the test cases. The existing correctness records will be removed.
- *Check*: Check the correctness of the test cases, by comparing the output fields against the selected metamorphism(s).
- *Analyse*: Invoke the selected analyser(s) to generate test report.

A distinctive feature of Morphy is its capability of test automation at the strategy level. Generally speaking, a test strategy is a rule that determines what, when and how testing activities should be performed. A typical example of a test strategy is the bottom-up integration testing strategy. In this strategy, software units that do not depend on other software units are integrated and tested first. Then, integration and testing moves up the dependence hierarchy gradually until the system is fully integrated. As far as we know, there is no report of successful automation of test strategies in the literature. They are executed manually in practice.

In the datamorphic testing model, test activities can be represented as invocations of test morphisms implemented as executable test code. Test strategies can, therefore, be defined as algorithms parametrised by test entities and test morphisms. Thus, they can also be implemented for automatic execution. This can significantly improve test efficiency and effectiveness, since the test morphisms can be combined in arbitrarily complex ways.

Morphy has implemented ten test strategies of three different kinds that target the requirements of testing AI applications. They are:

- *Mutant combination* strategies specify how to combine datamorphisms to generate mutant test cases. Details of these strategies is given in [13].
- *Domain exploration* strategies target the clustering and classification type of AI application. They employ various search algorithms to discover the borders between the subdomains between clusters/classes implemented by a machine learning model [7].
- *Test set optimization* strategies employ variants of genetic algorithms to optimize test sets through evolutionary computing.

Morphy enables the employment of these test strategies by selecting test morphisms and other parameters of the algorithms and execute them automatically.

Morphy also provides the test script facility to enable test automation at process level. Interactive operations of the Morphy test tools can be recorded and saved into a test script file. These can then be edited and replayed to automate the test process. The test script language can be found in [9].

3 Design And Implementation of The Morphy Agent

In this section, we present the design of the Morphy Agent, which has been implemented in Java. We will first present the architecture of the system, then presents two service request tools.

3.1 The Architecture

The overall structure of Morphy Agent, as illustrated in Fig. 3, is designed to allow a tester to perform testing by sending instructions and commands to test agents running remotely on a number of servers and to receive responses from these agents.

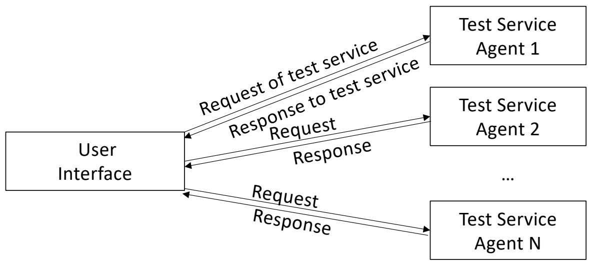


Figure 3: MorphyCloud Overall Structure

A tester can perform testing through a user interface running on his/her workstation to command a number of test service agents running on remote servers, for example, on cloud. To achieve the flexibility of implementing the user interface, the structure of test service agent is designed by applying the factory pattern of object oriented design show in Fig. 4.

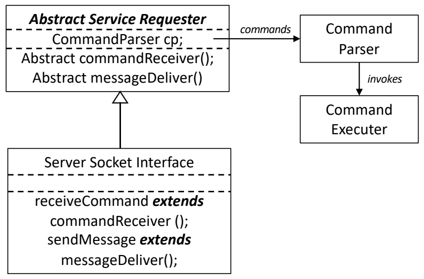

Abstract Service Requester
CommandParser cp;
Abstract commandReceiver();
Abstract messageDeliver()
commands
Command Parser
invokes
Command Executer
Server Socket Interface
receiveCommand extends commandReceiver ();
sendMessage extends messageDeliver();

Figure 4: Structure Of Test Service Agent

In Fig. 4, **ServiceRequester** is an abstract class that contains an attribute **cp** to link to the **CommandParser** and defines two abstract methods:

- **commandReceiver()**, which receives the service requests and sends to the **CommandParser** for executing;
- **messageDeliver()**, which delivers the service responses to the service requester.

The **CommandParser** parses the service request in the form of a string, parsing the string into the command and the parameters, and then invokes the corresponding function provided by **CommandExecuter** to perform a test activity or to execute a test strategy, etc. Its implementation of the abstract method **messageDeliver** sends service responses in the form of message to the requester through tcp/ip internet connection.

As shown in Fig. 5, the **CommandExecuter** wraps the components of Morphy so that its functions can be invoked flexibly. Consequently, Morphy Agent is highly compatible with Morphy such that test systems and test scripts that are executable on Morphy can also be executed on Morphy Agent.

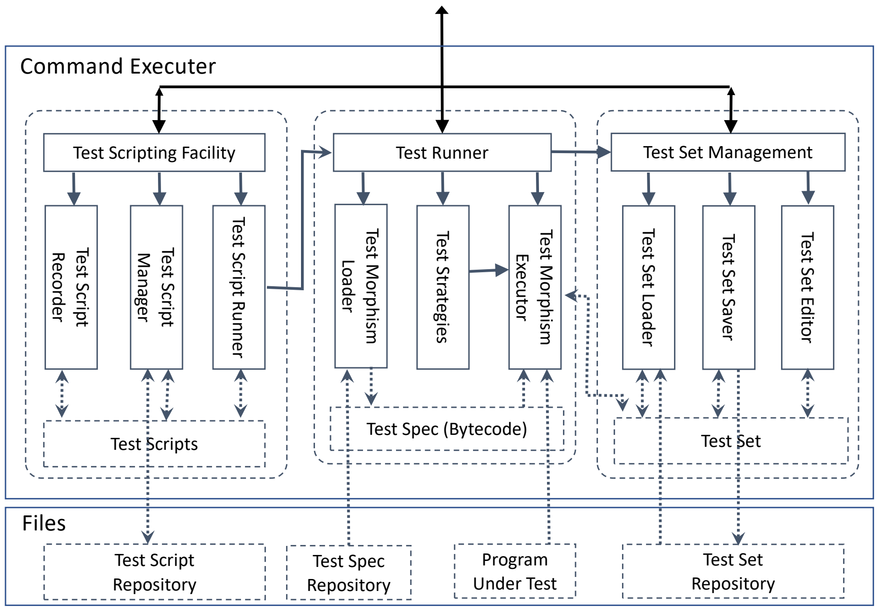


Figure 5: Structure of Command Executer

3.2 Command Line Interface

The **Command Line Interface** is a Java program, which receives a test request instruction from the command line that consists of two arguments for (a) the location of the **MorphyAgent** in the form of an IP address, and (b) the service request. It sends the service request to the Morphy agent on a machine at the IP address through the server socket on port 6129. It then listens to the input stream of the socket for the service responses and displays it on the screen.

The commands are in the format of:

$$MorphyCloud\ \ (-local| - remote@IpAddress)\ \ requestInstruction$$

The service requests are designed as one-to-one correspondence to the primary statements in Morphy's test script language, which is defined in [9]. These primary statements cover Morphy's functions described in Section 2.4.3.

Morphy's script language has five types of primary statements and a structured control statement. Every primary statement is a test request instruction of the same syntax and semantics as they are in Morphy's test script language. The structured control statement has no counterpart in the test service request instructions.

Primary statements include management statements, activity statements, strategy statements, communication statements and a script invocation statement. Their syntax and semantics are summarised in Table 2, where *mdName* is a test morphism method name, *mdNames* is a list of test morphism method names that starts with the square bracket symbol "[", separated by a comma "," and ends with the close square bracket symbol "]". In addition to instructions corresponding to these Morphy statements, Command Line Interface added a few new instructions for setting environment variables of the test service agent. Their syntax and semantics are also given in Table 2.

Table 2: The Syntax and Semantics of Test Service Request Instructions

Syntax	**Semantics**
Management Instructions	
loadTestSpec(fileName)	Load a Java .class file that implements a test system to the system
loadTestSet(fileName)	Load a test set file to the system
clean()	Clean the state of the system and re-initialising it
saveTestResult(fileName)	Save the current test set in the test suite to the file
Activity Instructions	
makeSeed(mdNames)	Invoke the specified seed maker(s) to make seed test cases
makeMutant(mdNames)	Invoke the specified datamorphism(s) to generate a set of 1'st order complete mutant test cases from the current set of test cases
measureTestSet(mdName)	Invoke the specified test set metric morphism
testSetFilter(mdNames)	Invoke the specified test set filter morphism(s) to transform the current test suite
executeTestCases()	Execute the test executer of the test system on all test cases in the current test suite
checkMetamorphisms(mdNames)	Invoke the specified metamorphism(s) to check the correctness of the program outputs and record correctness in the current test suite
analyse(mdNames)	Invoke the specified test result analyser morphism(s)
Strategy Invocation Instructions	
strategyName(parameters)	Apply the test strategy with the parameter; see Table 3 and 4 for details
Communication Instructions	
displayMessage(message)	Display a text message
saveMessage(fileName , message)	Write the text message to a file
Script Invocation Instructions	
playTestScript(fileName)	Invoke the test script stored in the file, which is determined by the *fileName* and the value of the environment variable *scriptFilePath*. When the value of *scriptFilePath* is the empty string, the *fileName* should be a string that represents an absolute file name. And, the execution of the instruction will set the value of *scriptFilePath* to be the file's parent; otherwise, the file will be get from the folder that the value of *scriptFilePath* variable represents without changing its value. The default value of *scriptFilePath* is the empty string when the system is initialised. The value of variable *scriptFilePath* can be set and reset through instructions that set environment variables.
saveTestScript(fileName)	Save the recorded test activities into the file, which is determined in the same way as *playTestScript* instruction.
Set Environment Variable Instructions	
startRecording	Start recording the test activities by appending the executed instructions to the test script records
stopRecording	Stop recording of test activities
setTestScriptPath(filePath)	Set the value of the variable *scriptFilePath* to be *filePath*
resetTestScriptPath()	Reset the value of the variable *scriptFilePath* to be the empty string
setFilePath(filePath)	Set the value of the variable *defaultFilePath* to be *filePath*
resetFilePath()	Reset the value of the variable *defaultFilePath* to be the empty string

The instructions for invoking test strategies and their corresponding parameters are given in Table 3. The meanings of the parameters are given in Table 4.

The following is an example of test script in Morphy's test script language for testing the trigonometric function $sin(x)$ in Java's Math library using the test system given in Section 2.3.

Table 3: Parameters of Strategy Invocation Instructions

Strategy Name	**Parameters**
firstOrderMutantCompleteStrategy	msNames; dmNames; tsfNames; mmNames
combinatorialCompleteStrategy	msNames; dmNames; tsfNames; mmNames
permutationCompleteStrategy	msNames; dmNames; tsfNames; mmNames
directedWalkExplore	dmNames
randomTargetExplore	NumOfPairs, dmName
randomWalkExplore	dmNames; Steps,NumOfStartPoints
GAOptimizeFilter	dmNames; msNames; tsfNames; tsmName, Steps, MutantRatio, ChangeRate
GAOptimizeWeakestRemoval	dmNames; msNames; tcmName, tsmName, Steps, MutantRatio, RemoveRatio, ChangeRate
GAOptimizeRandomRemoval	dmNames; msNames; tsmName, Steps, MutantRatio, RemoveRatio, ChangeRate
GAOptimizeStrongestMutant	dmNames; msNames; tcmName, tsmName, Steps, MutantRatio, RemoveRatio, ChangeRate

Table 4: Meaning of Parameters in Strategy Invocation Instructions

Parameter Name	**Meaning**
msNames	A list of seed maker names
dmNames	A list of datamorphism names
tsfNames	A list of test set filter names
mmNames	A list of metamorphism names
tcmName	The name of a test case metric
tsmName	The name of a test set metric
Steps	An integer value to control the maximal number of steps to travel or evolution cycles
MutantRatio	A real value between 0 and 1 to control the ratio of mutants over the test case pool to be generated each cycle of evolution in the genetic algorithm
RemoveRatio	A real value between 0 and 1 to control the ratio of test cases in the test set to be removed each cycle of evolution in the genetic algorithm
ChangeRatio	A real value between 0 and 1 to control the termination of the genetic algorithm, which will stop when the difference in test set's fitness between two consecutive evolution cycles is less than the ChangeRatio

Listing 4: Example of Morphy Test Script

```
loadTestSpec(file:C:\Morphy\examples\bin\SinTest.class)
makeSeed([randomValues])
makeMutants1stOrderComplete([piMinus,mid])
executeTestCases()
checkMetamorphisms([PiMinusAssertion])
MeasureTestSet([avgDistance])
```

The test script starts with loading the test system from a Java class file. Then, it invokes the seed maker **randomValues** to generate a set of 100 random real numbers in the interval $[0, \frac{1}{2} \times \pi]$. The datamorphisms **piMinus** and **mid** are then invoked to generate a set of mutant test cases which are then added to the test suite. Then the $sin(x)$ function is executed on all test cases. The correctness of the program on the test cases is checked against the metamorphism **PiMinusAssertion**. Finally, the quality of the test set is measured by invoking the test set metrics **avgDistance**. The code Listing 5 is the corresponding test script in Window's Batch shell scripting language for executing MorphyCloud test service agent on a server at the IP address **161.73.158.226**.

Listing 5: Example of Batch Test Script

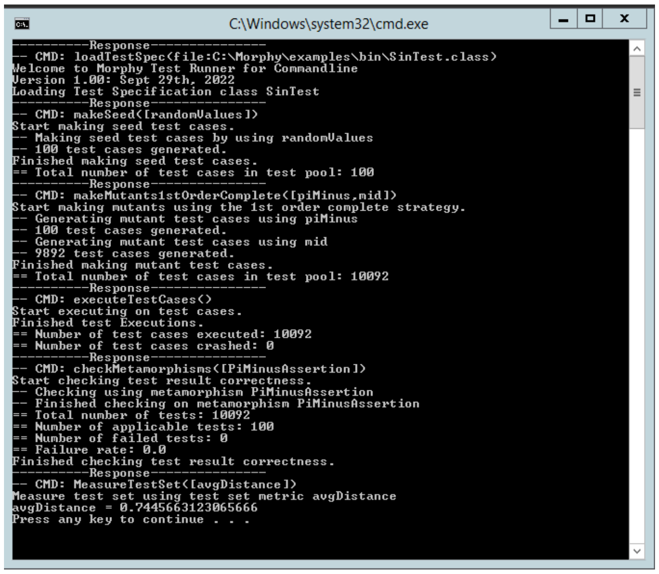
```
@echo off
cd C:\Morphy
set mc=java -jar MorphyCloud.jar
set location=-remote@161.73.158.226
%mc% %location% loadTestSpec(file:C:\Morphy\examples\bin\SinTest.class)
%mc% %location%  makeSeed([randomValues])
%mc% %location%  makeMutants1stOrderComplete([piMinus,mid])
%mc% %location%  executeTestCases()
%mc% %location%  checkMetamorphisms([PiMinusAssertion])
%mc% %location%  MeasureTestSet([avgDistance])
pause
```

Fig. 6 is the screen snapshot of the response messages received when running the Batch script to invoke the test service agent remotely.

The Command Line Interface presented in the above subsection are powerful for invoking and controlling Morphy test agents because complicated test scripts can be written in shell scripting languages and executed through operating system's commands. However, it has the drawback that shell scripts are not capable of controlling multiple agents to perform testing activities in parallel and distributed on a remote cluster of computers. Test tasks in such a test script is inevitably executing sequentially rather than in parallel because of their synchronous computing model. In the next subsection, we present a solution to this problem.

3.3 Text Interface

The **Text Interface** receives commands in the same form of **requestInstructions** from the keyboard and displays responses through concurrent threads in separate windows, where each window for one connected agent. It can be connected to a remote machine through the command **connect(IPAddress)**, then it reads user's input of commands from the keyboard, sends the control commands to the connected agent. After sent a command to the connected agent, it create a new thread to receive and display the responses from the agent, and then move on to receive and process the next command. In other words, it is asynchronous. Note that, when the Morphy test service agent is running on the local machine, the connect command can be written as **connect(local)**.



```
----------Response---------------
-- CMD: loadTestSpec(file:C:\Morphy\examples\bin\SinTest.class)
Welcome to Morphy Test Runner for Commandline
Version 1.00: Sept 29th, 2022
Loading Test Specification class SinTest
----------Response---------------
-- CMD: makeSeed([randomValues])
Start making seed test cases.
-- Making seed test cases by using randomValues
-- 100 test cases generated.
Finished making seed test cases.
== Total number of test cases in test pool: 100
----------Response---------------
-- CMD: makeMutants1stOrderComplete([piMinus,mid])
Start making mutants using the 1st order complete strategy.
-- Generating mutant test cases using piMinus
-- 100 test cases generated.
-- Generating mutant test cases using mid
-- 9892 test cases generated.
Finished making mutant test cases.
== Total number of test cases in test pool: 10092
----------Response---------------
-- CMD: executeTestCases()
Start executing on test cases.
Finished test Executions.
== Number of test cases executed: 10092
== Number of test cases crashed: 0
----------Response---------------
-- CMD: checkMetamorphisms([PiMinusAssertion])
Start checking test result correctness.
-- Checking using metamorphism PiMinusAssertion
-- Finished checking on metamorphism PiMinusAssertion
== Total number of tests: 10092
== Number of applicable tests: 100
== Number of failed tests: 0
== Failure rate: 0.0
Finished checking test result correctness.
----------Response---------------
-- CMD: MeasureTestSet([avgDistance])
Measure test set using test set metric avgDistance
avgDistance = 0.7445663123065666
Press any key to continue . . .
```

Figure 6: Screen Snapshot of The Responses of Executing The Batch Test Script

Fig. 7(b) and (c) shows two windows that display the responses from two service agents simultaneously while they are controlled by the Text Interface in (a).

The **Text Interface**'s most important difference from **Command Line Interface** is that it can directly execute test scripts written in Morphy's test script language through a new instruction **meta-playTestScript(fileName)**. When user enters the command, it reads the test script from the file stored on the tester's workstation where the (Text Interface) runs, and then sends the instructions in the script one by one to the currently connected agent. This new instruction is called a *meta instruction* because it controls the **Text Interface** itself rather than the connected agent. There are set of such meta instructions implemented in the **Text Interface**. They enable the tester to control multiple agents running on different machines at the same time and to execute test tasks on these machines in parallel. Table 5 gives the details of the meta instructions provided by the **Text Interface**.

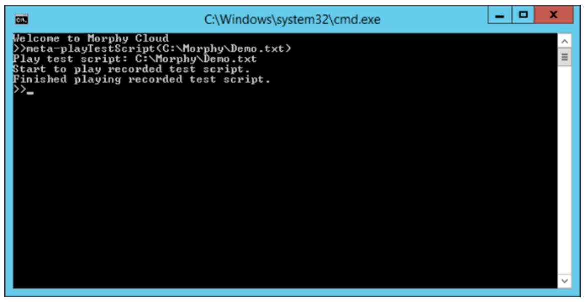
It is worthy noting that meta instructions can also be included in test scripts that are run on a **Text Interface** and invoked through the **meta-playTestScript** command. Such a test script is called a *meta test script*, while ordinary test scripts that are run by test service agents cannot include meta instructions. However, ordinary test scripts can also be run by a **Text Interface** through the **meta-playTestScript** instruction with the exactly same effect as run by a service agent via the instruction **playTestScript**. The code Listing reflst:MetaTestScriptExample1 shows a meta test script that instructs two MorphyCloud test service agents to execute the test script shown in Code Listing 4, which is stored in the file $C : /Morphy/SinTestScript.txt$ on their machines. The test script will be executed on two machines in parallel.

Listing 6: Example of Meta Test Script

```
connect(161.73.158.226)
playTestScript(C:\Morphy\SinTestScript.txt)
connect(161.73.158.227)
playTestScript(C:\Morphy\SinTestScript.txt)
```

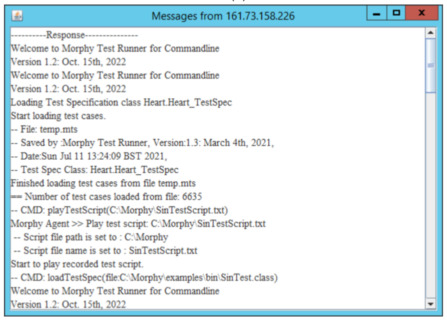
Table 5: The Syntax and Semantics of Meta Instructions

Syntax	**Semantics**
New Management Instructions	
connect(IPAddress)	Connect the control interface to the agent running on the machine at the IP address. When the IP address is 'local', it will connect to the Morphy test agent running on the local machine.
uploadFile(localFileName, remoteFileName)	Upload the file on the workstation to the remote machine that the controller is currently connected to and the save the data into a file with the *remoteFileName*. The locations of the files are determined by the environment variables *defaultFilePath* such that the value of the variable stored on the workstation determines the location of the local file and the value on the remote machine for the remote file.
Meta Script Invocation Instructions	
meta-playTestScript(fileName)	Command the agent to execute the test script stored in the file on the machine that the Text Interface runs, which is determined by the fileName and the value in the environment variable *scriptFilePath* of the interface. When the test script path is empty, the *fileName* should be a string that represents an absolute file name. And, the execution of the instruction will set the value of scriptFilePath to be the file's parent; otherwise, the file will be get from the folder that the scriptFilePath variable represents without changing its value. The default value of scriptFilePath is the empty string when the Text Interface is initialised. The variable scriptFilePath is an environment variable that can be set and reset through a set of meta instructions, which are instructions to control the Text Interface.
meta-saveTestScript(fileName)	Save the test activities recorded by the Text Interface into the file on its machine, which is determined in the same way as *meta-playTestScript* instruction.
Meta Environment Variable Operation Instructions	
meta-startRecording	Instruct the Text Interface to start recording the test activities by appending the executed instructions to the test script records
meta-stopRecording	Instruct the Text Interface to stop recording of test activities
meta-setTestScriptPath(filePath)	Set the value of the meta variable *scriptFilePath* of the Text Interface to be *filePath*
meta-resetTestScriptPath()	Reset the value of the meta variable *scriptFilePath* of the Text Interface to be the empty string
meta-setFilePath(filePath)	Set the value of the meta variable *defaultFilePath* of the Text Interface to be the parameter
meta-resetFilePath()	Reset the value of the meta variable *defaultFilePath* of the Text Interface to be empty



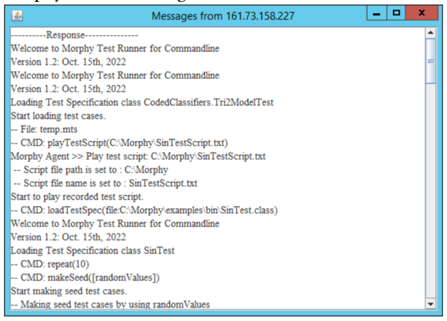
```
Welcome to Morphy Cloud
>>meta-playTestScript(C:\Morphy\Demo.txt)
Play test script: C:\Morphy\Demo.txt
Start to play recorded test script.
Finished playing recorded test script.
>>_
```

(a) Text Interface to Control Morphy Test Service Agents



```
----------Response---------------
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
Loading Test Specification class Heart.Heart_TestSpec
Start loading test cases.
-- File: temp.mts
-- Saved by :Morphy Test Runner, Version:1.3: March 4th, 2021,
-- Date:Sun Jul 11 13:24:09 BST 2021,
-- Test Spec Class: Heart.Heart_TestSpec
Finished loading test cases from file temp.mts
== Number of test cases loaded from file: 6635
-- CMD: playTestScript(C:\Morphy\SinTestScript.txt)
Morphy Agent >> Play test script: C:\Morphy\SinTestScript.txt
 -- Script file path is set to : C:\Morphy
 -- Script file name is set to : SinTestScript.txt
Start to play recorded test script.
-- CMD: loadTestSpec(file:C:\Morphy\examples\bin\SinTest.class)
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
```

(b)



```
----------Response---------------
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
Loading Test Specification class CodedClassifiers.Tri2ModelTest
Start loading test cases.
-- File: temp.mts
-- CMD: playTestScript(C:\Morphy\SinTestScript.txt)
Morphy Agent >> Play test script: C:\Morphy\SinTestScript.txt
 -- Script file path is set to : C:\Morphy
 -- Script file name is set to : SinTestScript.txt
Start to play recorded test script.
-- CMD: loadTestSpec(file:C:\Morphy\examples\bin\SinTest.class)
Welcome to Morphy Test Runner for Commandline
Version 1.2: Oct. 15th, 2022
Loading Test Specification class SinTest
-- CMD: repeat(10)
-- CMD: makeSeed([randomValues])
Start making seed test cases.
-- Making seed test cases by using randomValues
```

(c)

Figure 7: Screen Snapshots of The Text Interface

In contrast, the Batch script equivalent to this as shown in code Listing 7 will execute the test script on two machine sequentially. That is, the execution of the test script on the machine at IP address ”161.73.158.227” can only start after its execution on machine at IP address ”161.73.158.227” finishes.

Listing 7: Example of MorphyCloud Test Script in Batch Script

```
@echo off
cd C:\Morphy
set mc=java —jar MorphyCloud.jar
set location1=—remote@161.73.158.226
set location2=—remote@161.73.158.222
%mc% %location1% playTestScript(C:\Morphy\SinTestScript.txt)
%mc% %location2% playTestScript(C:\Morphy\SinTestScript.txt)
pause
```

4 Conclusion

In this paper we presented a test service agent that re-wraps Morphy’s functions. When test agents are running on a clusters of computers remotely, test instructions can be invoked and testing can be performed on a large volume of test data in parallel and distributed to a number of servers.

In this paper, we have also presented two interfaces that enable testers to send test service requests to and receive responses from the Morphy agents running on remote machines. The

Command Line Interface enhances the power of datamorphic testing methodology by enabling powerful test scripts written in any scripting languages while the core function of Morphy can be invoked as the basic commands of the scripts. The **Text Interface** on the other hand enhances the scalability of Morphy by introducing meta instructions and meta test scripts to control the distributed and parallel executions of test agents. Through these interfaces, a tester can flexibly operate a distributed system that consists of multiple test service agents.

For future work, we are further developing a graphic user interface to manage and operate a cluster of computers where test service agents are running in a user friendly environment.

References

[1] Y. Bagge. Experiments with testing a bounded model checker for C. MSc Dissertation, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK, Sept 2019.

[2] T. Y. Chen, F-C. Kuo, H. Liu, P-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, Jan 2018.

[3] S. Mugutdinov. Applying datamorphic technique to test face recognition applications. BSc dissertation, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford, UK, March 2019.

[4] L. Shan and H. Zhu. Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. *The Computer Journal*, 52(5):571–588, Aug 2009.

[5] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[6] H. Zhu. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In *Proc. of The 2nd Int'l Conf. on Trustworthy Systems and Their Applications (TSA 2015)*, pages 8–15, July 2015.

[7] H. Zhu and I. Bayley. Exploratory datamorphic testing of classification applications. In *Proc. of The 1st IEEE/ACM International Conference on Automation of Software Test (AST 2020)*, pages 51–60, July 2020.

[8] H Zhu and I Bayley. Discovering boundary values of feature-based machine learning classifiers through exploratory datamorphic testing. *Journal of Systems and Software*, 187:111231, May 2022.

[9] H. Zhu, I. Bayley, D. Liu, and X. Zheng. Morphy: A datamorphic software test automation tool. Technical Report OBU-ECM-AFM-2019-01, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford, UK, Dec. 2019.

[10] H. Zhu, I. Bayley, D. Liu, and X. Zheng. Automation of datamorphic testing. In *Proc. of 2nd IEEE International Conference on Artificial Intelligence Testing (AITest 2020)*, page In Press, May 2020.

[11] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29(4):366–427, Dec. 1997.

[12] H. Zhu, D. Liu, I. Bayley, R. Harrison, and F. Cuzzolin. Datamorphic testing: A methodology for testing ai applications. Technical Report OBU-ECM-AFM-2018-02, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK, Dec 2018.

[13] H. Zhu, D. Liu, I. Bayley, R. Harrison, and F. Cuzzolin. Datamorphic testing: A method for testing intelligent applications. In *Proc. of The First IEEE International Conference on Artificial Intelligence Testing (AITest 2019)*, pages 149–156, Los Alamitos, CA, USA, Apr 2019. IEEE Computer Society.