Supporting Test Suite Evolution through Test Case Adaptation

Mehdi Mirzaaghaei*, Fabrizio Pastore† and Mauro Pezzè* †
*Faculty of Informatics, University of Lugano, Switzerland
†Department of Informatics Systems and Communications, University of Milano - Bicocca, Milano, Italy
Email: {mehdi.mirzaaghaei,fabrizio.pastore,mauro.pezze}@usi.ch

***Abstract*—Software systems evolve during development and maintenance, and many test cases designed for the early versions of the system become obsolete during the software lifecycle. Repairing test cases that do not compile due to changes in the code under test and generating new test cases to test the changed code is an expensive and time consuming activity that could benefit from automated approaches.**

In this paper we propose an approach for automatically repairing and generating test cases during software evolution. Differently from existing approaches to test case generation, our approach uses information available in existing test cases, defines a set of heuristics to repair test cases invalidated by changes in the software, and generate new test cases for evolved software.

The results obtained with a prototype implementation of the technique show that the approach can effectively maintain evolving test suites, and perform well compared to competing approaches.

***Keywords*-Software testing, Test case evolution, Test case maintenance**

I. Introduction

Software systems evolve both during development and maintenance. Modern processes postulate incremental development and testing first, while maintenance activities incrementally modify the software systems usually available with test cases. Incremental changes in the code may invalidate some test cases, and developers need to both correct test cases that are invalidated by the software changes, and add new ones to test new functionality. We use the term *test suite evolution* to indicate the problem of automatically evolving an original set of test cases by either repairing test cases invalidated by the changes in the code or generating test cases to test new functionalities.

Evolving test cases is an expensive and time consuming activity, and sometime developers discontinue test cases that may be reused, and do not generate all test cases that may be needed, due to time pressure [1]. Automated approaches reduce the pressure on developer, and increase the amount of test cases reused along the development and maintenance process.

So far, researchers have looked at the problem of automating testing activities from different viewpoints. Regression testing focuses on selecting subsets of test cases to reduce the amount of test cases to be re-executed [2]. Regression testing techniques are useful when dealing with large test suites, but do not address the problem of obsolescence of test cases that are invalidated by relatively small changes in the software, like changes that derive from bug fixing, refactoring and incremental development activities.

Automatic test case generation techniques derive test cases either from models [3] or code [4], [5]. They tend to generate test cases that may be both incomplete and difficult to understand. These techniques often do not identify the setup actions necessary to execute the test cases, and generate a large amount of test inputs without distinguishing between valid and invalid inputs, thus causing invalid failures. Techniques for mining source code alleviate but do not eliminate the problem [6]. Most of the existing techniques do not generate test oracles, thus forcing developers to inspect the generated test cases to write oracles [7].

Automatic test case repairing techniques are in their infancy. Modern development frameworks like Eclipse[1] can identify simple syntactic fixes that can be useful, but address only few elementary problems. Marinov et al. go beyond simple fixes proposing techniques to correct oracles for interactive applications [8].

In this paper, we introduce a new approach that reuses information available in existing test cases to automatically evolve test suites. In particular, we propose a set of algorithms that can automatically evolve test suites.

We manually inspected different versions of open source systems, and detected that software developers often reuse and adapt existing test cases to evolve test suites. We identified frequent *actions for adapting test cases* that developers commonly adopt to correct and generate test cases, and we relied on the identified actions to define *algorithms for evolving test cases* as a solution to support software developers in the evolution of test cases.

This paper contributes to the state of the art by:

- identifying a set of *actions for adapting test cases* commonly adopted by software developers;
- defining *TestCareAssistant* (*TCA*), a framework that automate test suites evolution by implementing different algorithms for evolving test cases;
- evaluating TCA with state of the art techniques like Randoop [9], Google CodePro AnalytiX[2] (CodePro),

[1]http://www.eclipse.org
[2]Google CodePro AnalytiX, http://code.google.com/javadevtools/codepro

and EvoSuite [10] when applied to generate test cases for more than 300 classes under test.

The paper is organized as follows. Section II summarizes related work. Section III illustrates *TCA*. Sections IV and V illustrate in details two of the algorithms, *RepairSignatureChanges* and *TestClassHierarchies*, that we implemented in the *TCA* prototype. Section VI presents some empirical results that show the effectiveness and limits of our approach. Section VII concludes discussing the main results achieved so far and outlining our current research plans.

II. Related work

In modern software development and maintenance approaches, test cases are produced early in the process and executed many times. Changes in software may invalidate test cases and require new ones. Regression testing focuses mainly on selecting subsets of large test suites to reduce the cost of execution [2]. In this paper we focus of the problem of *evolving test suites* that we define as the problem of correcting test cases that become obsolete during software evolution and generating new test cases to test new functionalities. So far, this problem has been approached with either test case generation or repair techniques.

Test cases are generated automatically either from specifications or code. Techniques for automatically generating test cases from specifications, also called model-based testing, transform models into sets of test cases [3]. These techniques benefit from model-driven development, but require the availability of complete models of the system.

Approaches to generate test cases from code rely on symbolic execution, search techniques or random testing. Symbolic execution techniques identify the inputs that cover the feasible elements of a program by solving the path conditions of the program [4]. Although modern tools that generate test cases by solving path conditions, like CodePro, find industrial applicability for generating test cases, symbolic execution works well when conditions involve simple local variables, but do not scale with the complexity of the system, when conditions involve dynamically allocated variables and the length of the paths produces big expressions hard to evaluate. Concolic execution reduces the constraint solving limitations by combining path constraints evaluation with random generation of test inputs [11]. Other approaches like Randoop [9] completely remove limitations of constraint solving by generating input values randomly. Search-based testing techniques, like Evosuite [10], generate test cases by combining search techniques, for example genetic algorithms, with ad-hoc objective functions that maximize testing goals, for instance branch coverage.

Automatic test case generation techniques usually do not identify the setup actions necessary to execute the test cases, and tend to generate a huge amount of test cases without distinguishing among valid and invalid inputs thus causing invalid failures. Furthermore automatically generated test inputs are often hard to read and maintain, and their practical applicability is limited to regression testing or detection of unexpected exception conditions [1].

Only recently, some researchers started investigating test case generation techniques that produce readable and maintainable test cases. Robinson et al. generate maintainable test cases by extending Randoop with actions derived from field experience [1], while Mseqgen generates readable test inputs by mining source code of programs built upon current program under test [6].

Techniques that automatically repair test cases rely on models and source code. Memon et al. compare models of GUI events to capture the differences between two software versions, and update GUI test cases accordingly [12]. The technique works well only in the presence of application models and runtime assertions. Test augmentation techniques instead do not rely on models but use symbolic execution [13], concolic execution or genetic algorithms [14] to derive test inputs that cover paths not executed yet. These approaches improve automatic test case generation techniques by taking advantage of the test cases manually written by developers, but suffer from the same limitations of classic automatic techniques when generating new test inputs. Furthermore test augmentation techniques do not reuse existing information to test new functions or classes, but fully rely on test case generation approaches.

ReAssert focuses on repairing test oracles, and repairs oracles broken by changes in the software specifications by re-executing the failing test cases by means of concrete or symbolic execution [8]. The suggestions proposed by ReAssert must be validated by software developers, since ReAssert modifies test cases to make them pass, and thus it could erroneously mask failures. ReAssert repairs test oracles, but does not fix test inputs, thus developers still need to manually correct the inputs.

Some test cases can also be corrected by means of automatic refactoring techniques that can prevent simple errors by automating some of the possible refactoring activities like moving or renaming methods. Unfortunately common refactoring practices like adding new parameters to methods [15] are only partially automated by existing tools and techniques. For example, ReBa [16] and Eclipse can fix compilation errors caused by parameter changes only when the modified parameters can be replaced by default values. Other approaches mine source code to repair invalid method invocations but require that software developers manually adjust method call arguments [17] or validate the suspicious method calls identified [18].

III. Test Evolution Algorithms

In this paper we present *TCA*, a framework that supports software developers in test cases maintenance by automating the activities commonly adopted by software developers to maintain test cases. We empirically investigated how developers correct obsolete test cases and reuse existing ones to generate new test cases on three different open source systems: JFreeChart[3] a graph plotting application, PMD[4] a source code analysis toolset, and JodaTime[5] a library for date time format management. So far, we identified five test adaptation approaches commonly adopted by software developers to evolve test suites, and defined corresponding algorithms that comprise *TCA*.

TCA requires four inputs from software developers: the original and the modified version of the program, the test cases written for the original program and the name of the test case to repair or the class to generate test cases for. *TCA* evolves the test suite by:

1) Analyzing the software changes by diffing the original and modified version of a software;
2) Adapting the test cases using the appropriate test evolution algorithms.

Finding differences between source code (step 1) is a well addressed problem, and there is a lot of reliable tool support. In this section, we focus on the problem of adapting test cases (step 2), and informally introduce the algorithms for evolving test cases.

Algorithm 1: Repair Signature Changes. Software developers often change declaration of method parameters and return values, and need to repair the compilation errors induced by the changes. *TCA* repairs the test cases by changing the initialization of the variables used in the test case. It first diffs the original and modified software to identify the type of the change, for instance, parameter type change, and determines the type that must be used in the modified test case to repair the compilation error. *TCA* then identifies common paths in the data flow of the modified parameters, and determines the values used in the original test cases that can be reused to repair the test case while preserving the test behavior.

Algorithm 2: Test Class Hierarchies. In object oriented systems, developers often add software functionalities by extending class hierarchies. Whenever developers introduce a new class in a hierarchy, they define new test cases to verify the functionality of the new class. *TCA* automatically generates test cases for the new class by copying and adapting the test cases for the existing classes of the hierarchy. *TCA* identifies similar classes, usually the ancestor and sibling classes, and reuses the test cases available for those classes. It first replaces references to the original class under test with the new class, then resolves compilation errors by updating references to fields, and finally uses algorithm 1 to modify the initialization of the variables used in the method invocations to match the signature of the methods in the new class. The oracles of the original test cases may not work for the modified test cases because the two classes implement different functionalities. *TCA* adapts the oracles of the original test cases by identifying the assertions that fail, and uses *ReAssert* to replace the expected values with the actual values returned by the class under test. This approach leads to valid oracles only if the implementation is correct, thus developers should validate the updated oracles. Verifying updated oracles is cheaper than manually replacing the values of all the oracles in a test.

Algorithm 3: Test Interface Implementations. The test cases developed to verify two classes that implement the same interface usually share a common behavior but may differ in terms of setup actions and oracles, for example constructors may require different types of parameters. *TCA* generates test cases for a new implementation of a given interface by copying and adapting the test cases developed for the classes that implement the same interface. First, *TCA* removes compilation errors by redefining the variables passed as inputs to setup methods with proper input parameters. Then *TCA* identifies failing oracles and updates their expected values following an approach similar to the one adopted in Algorithm 2.

Algorithms 4: Test New Overloaded Methods. Overloaded methods share their name but not the number and type of their parameters, and usually differ slightly in the implemented functionality. *TCA* generates test cases for a new overloaded method by copying and adapting the test cases developed for other overloaded methods. *TCA* first updates the input parameters passed to the method under test according to the same strategy adopted for Algorithm 1, and then updates the result expected by the oracle assertions.

Algorithm 5: Test New Overridden Methods. When developers override a method, they generate a new method with the same signature of a method defined in a parent class. Overridden methods share the same interface but differ for the results they generate. *TCA* generates the test cases for a new overridden method by copying the test cases developed for the method of the parent class and changing the subjects of the test: It substitutes the instances of the parent class used in the original test case with instances of the child class. *TCA* reduces the developers effort in updating oracles by automatically updating the expected values of the oracle assertions to reflect the overridden method implementation.

We implemented *TCA* as an Eclipse plug-in that currently implements algorithms 1 and 2. Sections IV and V detail the two algorithms.

IV. Repair signature changes

The algorithm *RepairSignatureChanges* repairs test cases broken by changes in method signatures. It does not target the compilation errors that are successfully repaired by the refactoring tools provided by popular IDEs like method renaming and changes in the parameter order. The algorithm also ignores modifications that do not cause compilation

[3]http://www.jfreechart.org
[4]http://pmd.sourceforge.net/
[5]http://joda-time.sourceforge.net/

errors, for example the introduction of a return value. *RepairSignatureChanges* focuses on four types of changes in the method signature: changing the type of one or more parameters, removing one or more parameters, adding one or more parameters, changing the return type. The empirical results discussed in Section VI-A, show that these changes are common and may invalidate a lot of test cases.

Figure 1 shows an example of a change in a method declaration that breaks some test cases of PMD. During the development of version 1.1 of PMD, developers modified the signature of method `Report.addRule`. The change caused compilation errors in the 13 test cases that use the modified method. Figure 2.a shows one of the broken test cases: the change causes a compilation error in line 3 because the test case uses the variable `filename`, which is of type `String`, as second parameter of the method `Report.addRule`, but the method `Report.addRule` requires a parameter of type `Context` in version 1.1 of PMD.

RepairSignatureChanges can address multiple compilation errors, and copes with each of them iteratively. At each iteration, it adapts the test cases by (1) analyzing the change that caused the compilation error, (2) determining the initialization values that shall be used to repair the compilation error, and (3) repairing the compilation error.

A. Analyzing the Change

When invoked after a compilation error, *RepairSignatureChanges* identifies both the changed elements (the modified parameters or return values), and the type of change.

RepairSignatureChanges identifies the methods that need to be repaired using the information about the compilation errors. If a compilation error does not involve a method invocations, *RepairSignatureChanges* cannot fix the error and returns. Otherwise, *RepairSignatureChanges* identifies the type of the change by diffing the signatures of the modified methods. In the current prototype, we target Java programs, and use JDiff[6], which returns the set of elementary actions (parameter additions, removals, type changes, and return type changes) that correspond to the differences between the original and the modified methods. When applied to the PMD example, *RepairSignatureChanges* determines that the type of the second input parameter of method `addRule` has been changed from `String` to `Context`.

a) PMD 1.0
```
1 void addRule(int line, String file){
2   this.line = line;
3   this.name = file; }
```
b) PMD 1.1
```
1 void addRule(int line, Context ctx){
2   this.line = line;
3   this.name = ctx.getFilename(); }
```

Figure 1. A change of method Report.addRule in PMD v 1.1. Developers changed the type of the second parameter from `String` to `Context`.

[6]http://www.jdiff.org/

a) Original test case
```
1 Report r= new Report();
2 String filename="foo";
3 r.addRule(5, filename);
4 assertTrue(!r.isEmpty());
```
b) Repaired test case
```
1 Report r = new Report();
2 String filename ="foo";
3 Context ctx = new Context(filename);
4 r.addRule(5, ctx);
5 assertTrue(!r.isEmpty());
```

Figure 2. a) A test case broken by the change in Figure 1: type mismatch causes a compilation error in line 3. b) The test case repaired by *TCA*.

B. Determining the Initialization Values

After identifying the elementary change that caused the compilation error, *RepairSignatureChanges* determines first the program variables that must be initialized, and then the proper initialization values that preserve the behavior of the test case. We describe these two steps considering the case of parameter modification. The algorithms for parameter additions, removal and return type change are similar. The interested reader can find additional information in [19].

Determining the proper values to use to initialize the modified variables is hard, since object parameters can be complex to initialize, due to the presence of many attributes whose initialization values may be difficult to determine. However, not all the elements of an object may need to be initialized to execute a given test case. *RepairSignatureChanges* identifies the fields that must be initialized to execute the test cases and focuses on those only.

To identify the *variables to initialize*, *RepairSignatureChanges* starts by locating the *first use* of the modified parameter in the new version of the software system by means of static data flow analysis. If the first use is not unique, *RepairSignatureChanges* selects the one occurring first in the source code. If the modified parameter is an object, *RepairSignatureChanges* determines the scope of the required initialization by checking if the first use involves the whole object or a subset of its fields. If the first use does not involve the whole object, *RepairSignatureChanges* identifies the object attributes used during the execution of the method, and locates their first use. *RepairSignatureChanges* uses *Datec* [20] to identify all the uses of the attributes, and then traverses the interprocedural control flow graph [21] provided by *Soot*[7], a static analysis framework, starting from the modified method, to identify the first use of each attribute. If the modified parameter is either a primitive type or an object whose fields are not accessed in the first use, *RepairSignatureChanges* identifies the parameter itself as the only *variable to initialize*.

In the PMD example of Figure 1, *RepairSignatureChanges* determines that the parameter `ctx` is used in line 3 of the method `addRule` to invoke the method `Context.getFilename`. Thus, *RepairSignatureChanges* needs to determine the values to initialize the parameter `ctx`. *RepairSignatureChanges* locates all the uses

[7]http://www.sable.mcgill.ca/soot/

of the fields of class `Context` that are reachable from the invocation of method `addRule`. In this example, only the field `filename` is accessed within the execution of method `addRule`, since it is read by the getter method `getFilename`. Thus, *RepairSignatureChanges* needs to initialize only the field `filename`.

RepairSignatureChanges determines the proper initialization values –the values that preserve the test behavior– by looking for corresponding values used in the test cases of the original version of the software system.

To determine the initialization values, *RepairSignatureChanges* identifies the set of locations that correspond to uses of the *variables to initialize* in the modified software. *RepairSignatureChanges* includes into this list also the variables that are an exact copy of the *variables to initialize*. In the PMD example, *RepairSignatureChanges* finds two locations, the use of field `filename` in the getter method `getFilename` and the use of the value returned by method `getFilename` in line 3 of method `addRule`. *RepairSignatureChanges* then sorts the locations corresponding to the uses following the order in which *Soot* visits them while traversing the interprocedural control flow graph of the modified method.

For each *variable to initialize*, *RepairSignatureChanges* starts from the first use in the set and looks for a *corresponding line* in the original software. *RepairSignatureChanges* identifies the *corresponding line* by diffing the original and the modified source code of the method with *JDiff*. A line $L0$ of the original source code *corresponds* to a line $L1$ of the modified code if $L0$ is the counterpart of $L1$, as usually determined by the *Unix diff* algorithm, and $L0$ and $L1$ differ at most for: the name of the defined variable, and a literal or a variable that replaces the *variable to initialize* in the original software (we call this the *corresponding term*). If the *Unix diff* identifies multiple counterparts for line $L1$, *RepairSignatureChanges* selects the line most similar to $L1$ according to the Levenshtein distance [22]. If no corresponding line is found, *RepairSignatureChanges* proceeds with the next use, otherwise *RepairSignatureChanges* finds a term that corresponds to the use of the *variable to initialize* by applying the Needleman–Wunsch [23] global alignment algorithm on the line in the modified software and its corresponding in the original one. If *RepairSignatureChanges* does not find any corresponding line for a *variable to initialize*, it indicates the initialization value of this element as *unknown*. In the PMD example, line 3 of Figure 1.a is the line that *corresponds* to the use of `ctx.getFilename()` at line 3 in Figure 1.b, and `file` is the *term* that corresponds to field `ctx.filename`.

We can determine the value of the *corresponding term* in the original software either statically or dynamically. We described the static approach in a preliminary paper [24]. Here we describe the dynamic approach that is currently implemented in *RepairSignatureChanges*, and that overcomes the limitations of the static one. *RepairSignatureChanges* implements the dynamic approach by instrumenting the binaries of the original software with *Soot*, and by running the original version of the test cases to record the dynamic value of each *corresponding term*. At this stage *RepairSignatureChanges* has identified the variables to initialize and instruments only the locations corresponding to the uses of these variables. In the PMD example the execution of the instrumented version of the software allowed *RepairSignatureChanges* to determine that the value to be used to initialize field `filename` of class `Context` is the String ``` ``foo'' ```.

C. Repairing the Compilation Error

RepairSignatureChanges repairs the test case by first removing the compilation error and then initializing the test variables with proper values to preserve the test behavior. It removes the compilation error by substituting the original variables with variables of a proper type. In the case of parameter type change of the PMD example, *RepairSignatureChanges* behaves like software developers by defining a new variable of type `Context` (variable `ctx` in line 3 of Figure 2.b), and passing it as argument of the method `addRule` (line 4 of Figure 2.b).

RepairSignatureChanges then initializes the variables declared in the test cases according to the results of the previous steps. *RepairSignatureChanges* initializes the primitive parameters by simply assigning the values as computed in the phase *determine the initialization values*. It initializes the fields of the introduced objects by invoking the constructor that initializes most of the fields. As shown in Figure 2.b in the PMD example, *RepairSignatureChanges* initializes the variable `ctx` by instantiating an object of type `Context` using the constructor that initializes the field `filename`.

When the *variable to initialize* does not belong to the set of variables introduced by *RepairSignatureChanges*, or when the constructor of a variable introduced by *RepairSignatureChanges* does not initialize all the fields, *RepairSignatureChanges* invokes the setter methods that initialize these fields, or uses reflection if setter methods are not available.

When dealing with a return type change, *RepairSignatureChanges* creates a new variable, assigns the return value of the modified method to the new variable, and initializes the variable with the sequence of getters identified in the phase *determine the initialization values*.

V. Test Class Hierarchies

The algorithm *TestClassHierarchies* automatically generates the test cases for new classes added to a hierarchy.

Classes belonging to the same hierarchy share common interfaces and behaviors, and differ for some of the offered functionality. Software designers take advantage of these characteristics when developing test cases that often share setup actions, for example the objects under tests are built by passing the same parameters, present same invocation sequences, and use oracles that inspect the same output values, but expect different results.

```
48  private static final Chronology ETHIOPIC_UTC =
      EthiopicChronology.getInstanceUTC();
301 DateTime epoch = new DateTime(1, 1, 1, 0, 0, 0, 0, ETHIOPIC_UTC);
302 long millis = epoch.getMillis();
303 long end = new DateTime(3000, 1, 1, 0, 0, 0, 0, ISO_UTC).getMillis();
307 DateTimeField monthOfYear = ETHIOPIC_UTC.monthOfYear();
308 assertEquals(EthiopicChronology.EE, epoch.getEra());
316 while (millis < end) {
320     int monthValue = monthOfYear.get(millis);
324     if (monthValue < 1 || monthValue > 13)
325         fail("Bad month: " + millis);
330     assertEquals("EE", era.getAsText(millis));
341     // test leap year
342     assertEquals(yearValue % 4 == 3, year.isLeap(millis));
```

Figure 3. A test case for class `EthipicChronology`.

Figure 3 shows a test case for the class `EthiopicChronology` that belongs to JodaTime, and was added to the hierarchy of class Chronology in version 1.2 (revision 911). *TestClassHierarchies* generated the test case shown in Figure 3 by adapting the test case `testCalendar` developed for class `CopticChronology`. The test case iterates over all the days in the range 0 AC - 3000 AC, converts the day representation from ISO calendar to Ethiopic calendar, and then checks if the conversion is correct. A valid test case for class `EthiopicChronology` must properly combine different objects and methods. The function derives a valid month value in the Ethiopic calendar by retrieving the `DateTimeField` object that holds the month value (`monthOfYear` in line 307) from a `Chronology` object configured using an instance of class `EthiopicChronology` (`epoch` in line 301). The `DateTimeField` object gives the month in the Ethiopic Chronology that corresponds to a given timestamp (the conversion is done by invoking method `get`, line 320). Proper checks must be added, for example for the values of months and leap years (lines 324 and 342). *TestClassHierarchies* can generate all the elements of a valid test case, while a test case with random invocations of methods of class `EthiopicChronology` would not be meaningful. *TestClassHierarchies* can derive a lot of the domain information required to build good test cases from existing test cases, while competing techniques cannot. In the following, we illustrate the three steps that comprise the *TestClassHierarchies* approach.

A. Selecting and copying the test cases

TestClassHierarchies automatically identifies the candidate test cases by selecting all the test cases of the classes in the same hierarchy of the new class. All the classes in a hierarchy might share a common behavior, and thus their test cases can be used to generate new test cases for the classes in the hierarchy. *TestClassHierarchies* copies and renames the candidate test cases to prevent classpath conflicts.

B. Adapting candidate test cases

For each candidate test case, *TestClassHierarchies* first updates all the references to the class under test, and then solves compatibility issues raised by compilation errors. In the case of the test for class `EthiopicChronology` in Figure 3, *TestClassHierarchies* updates all the occurrences of term `CopticChronology` with term `EthiopicChronology`. To improve readability, *TestClassHierarchies* updates also all the variable names that contain portions of the name of the original class under test with portions of the name of the new class under test (we partition names according to the Java camel case convention). In the example of Figure 3, *TestClassHierarchies* replaces `COPTHIC_UTC` in lines 48 and 307 with `ETHIOPIC_UTC`. *TestClassHierarchies* identifies the class under test using a simple heuristic based on the standard naming of JUnit test cases for Java: It removes the prefix/-suffix "Test" from the name of the candidate test class.

The generated test cases may lead to compilation errors because of undefined fields, constant, constructors, or methods. In the following paragraphs, we describe how *TestClassHierarchies* addresses each incompatibility.

Adapting undefined fields and constants: Test cases often contain references to constants or fields declared in the classes under tests. *TestClassHierarchies* updates references to constants and fields of the original class under test by replacing them with references to the corresponding constants and fields declared in the class to test. It identifies corresponding fields and constants by applying the algorithm to find corresponding terms discussed in Section IV-B. For example when adapting the test case in Figure 3, *TestClassHierarchies* replaces the references to `CopticChronology.AM` with references to `EthipicChronology.EE` (see line 308), the corresponding constants used to indicate the default era.

```
g = new GraphicsMock();
output = new SVGOutput(
   g,
   DefaultEnvironment.DEFAULT_FONT, fgColour, bgColour);
```

```
output = new SVGOutput(
   new JavadocEscapeWriter(EasyMock.createMock(Writer)),
   DefaultEnvironment.DEFAULT_FONT, fgColour, bgColour, 0, "" );
```

Figure 4. Example of constructors call adaptation.

Adapting undefined constructors: Constructors of classes that belong to a same hierarchy may differ in the number and type of their parameters, although they may share a subset of their parameters.

Figure 4 shows the excerpt of a test case for class `SVGOutput` of Barbecue[8] v1.5 that

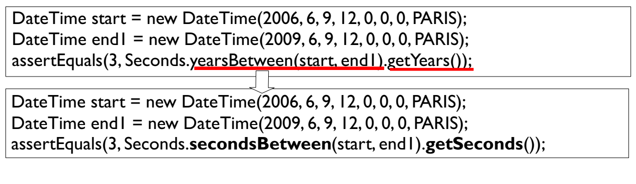
[8]Barbecue is a barcode Java library, http://barbecue.sourceforge.net/

TestClassHierarchies generated by adapting the test cases for class `GraphicsOutput`. The constructor of class `GraphicsOutput` used in the candidate test case receives as input four objects of type `Graphics2D`, `Font`, `Color` and `Color`, in this order. Class `SVGOutput` does not provide a constructor that receives the same input types, thus the candidate test case causes a compilation error.

To repair these errors, *TestClassHierarchies* replaces the original constructor calls with the constructor of the class under test that is most similar to the replaced one. *TestClassHierarchies* ranks all the constructors of the class under test on the basis of the parameter types shared with the constructor of the original class and the number of parameters with the same name of the sibling constructor. *TestClassHierarchies* selects the constructor with the highest rank.

After finding a similar constructor, *TestClassHierarchies* looks for parameters shared by the constructor used in the original test and the selected constructor. The matching is done by iteratively looking for parameters of the two constructors with the same type or name, with priority to types. *TestClassHierarchies* reuses the compatible parameters of the original test case by suitably positioning them in the new constructor call.

In the case of mismatching parameters, *TestClassHierarchies* generates input values as follow. If the parameter is primitive *TestClassHierarchies* uses default values: "0" for numeric types and bytes, the space character for the `char` type, the empty string for type String, and an array with a default element for type array. If the required parameter is an object, *TestClassHierarchies* uses either constants of that type or factory methods that return an object of that type. If none is found, *TestClassHierarchies* invokes a constructor of the required type[9]. If the constructor requires object parameters, *TestClassHierarchies* simply creates stubs. Our Java implementation uses Easymock[10]. Figure 4.b shows how *TestClassHierarchies* creates the first parameter of `SVGOutput` constructor, which is of type `Writer`: It invokes the constructor of class `JavadocEscapeWriter`[11], with a stub created using Easymock.



```
DateTime start = new DateTime(2006, 6, 9, 12, 0, 0, 0, PARIS);
DateTime end1 = new DateTime(2009, 6, 9, 12, 0, 0, 0, PARIS);
assertEquals(3, Seconds.yearsBetween(start, end1).getYears());
```

```
DateTime start = new DateTime(2006, 6, 9, 12, 0, 0, 0, PARIS);
DateTime end1 = new DateTime(2009, 6, 9, 12, 0, 0, 0, PARIS);
assertEquals(3, Seconds.secondsBetween(start, end1).getSeconds());
```

Figure 5. Example of repair of method invocations. *TestClassHierarchies* repairs the compilation error caused by the invocation of method `yearsBetween` by invoking the corresponding method, `secondsBetween`.

[9]In the case of interfaces, it randomly picks up a constructor that implements the interface

[10]http://easymock.org/

[11]Class `JavadocEscapeWriter` implements interface `Writer`

Adapt undefined method: Figure 5 shows a test case for class `Seconds` that *TestClassHierarchies* generated by reusing the test case for class `Years`. The test case causes a compilation error because method `yearsBetween(DateTime,DateTime)` is not declared in the new class under test.

When *TestClassHierarchies* identifies a call to a non existing method, looks for a similar method in the class under test. *TestClassHierarchies* finds the most similar method considering the signatures of all the methods declared in the class under test, and applies an extended version of the algorithm that identifies similar constructors: The algorithm first sorts the methods according to the similarity of their names by calculating their Levenshtein distance, and then according to the number of common parameter types and names. After identifying the most similar method, *TestClassHierarchies* replaces the invocation of the original method with the new one following the same steps adopted in the case of constructors: It adds corresponding parameters, and then looks for new parameters. If *TestClassHierarchies* does not find any similar method it removes the call to the undeclared method.

C. Repairing runtime errors

After repairing compilation errors, *TestClassHierarchies* executes the test cases (*TestClassHierarchies* discards the test cases that were not successfully repaired). The execution of the test cases might lead to pass, fail, or exception.

Test case failures may depend on the fact that oracles do not reflect the specifications of the class under test. *TestClassHierarchies* repairs the failing assertions with ReAssert [25]. ReAssert automatically updates JUnit assertions by replacing the expected value with a new value that is equal to the one returned by the method under test, and thus makes the test case pass. Identifying and fixing wrong oracle values is easier than writing new oracles, and we expect that software developers can easily check the oracles.

Test cases that throw an exception are a special case of failing test cases. Exceptions may either indicate an error in the implementation of the class, wrong test inputs, or an invalid test case setup. When the test case execution raises an exception, we rely on the developers to determine if the test cases should be removed or kept because they pinpoint a fault.

TestClassHierarchies generates test cases by reusing test cases written for different classes, and thus multiple test cases might cover the same software behavior. To prune duplicate test cases, *TestClassHierarchies* adopts a simple heuristic that consists of executing the test cases and measuring the instructions covered during execution[12]. *TestClassHierarchies* discards the test cases that do not increase the instructions coverage, i.e. that do not cover instructions not already covered.

[12]The current prototype uses EclEmma, www.eclemma.org

Subject	#	avg. LOC	Not-Compile	TCA	%
Cassandra	19	62,631	197,047	36,881	18.72
Geronimo	4	208,216	30,057	16,222	53.97
POI	6	236,614	47,502	16,723	35.20
Santuario	4	11,185	8,532	590	6.92
Shindig	6	96,813	3,708	2,570	69.31
Velocity	15	45,576	1,442	676	46.88
Wicket	7	203,183	6,930	6,246	90.13
Xindice	9	55,821	13,072	5,444	41.65
XMLBeans	7	159,835	3,922	1,986	50.64

Table I
CHANGES IN APACHE PROJECTS

VI. EXPERIMENTS

We evaluated the applicability and effectiveness of *TCA* on some open source projects. The current prototype allows us to experiment with the two algorithms described in the former sections, *RepairSignatureChanges* for repairing test cases to cope with signature changes, and *TestClassHierarchies* for generating new test cases for classes that extend class hierarchies. For both algorithms, we evaluated the ability of *TCA* to reduce the effort on test case maintenance, and its effectiveness in adapting the test suite. We report the results for the two algorithms in the following subsections.

A. Repairing signature changes

Applicability: To evaluate the applicability of *RepairSignatureChanges* we investigated how often *RepairSignatureChanges* can address changes that cause compilation errors in test cases. We analyzed 80 versions of different open source projects of the Apache Software Foundation[13]. We choose software projects with several versions properly tagged on the versioning repository. Table I reports the projects that we analyzed in column *Subject*, the number of versions that we considered in our study in column *#*, and the average size (lines of code) of each in column *avg. LOC*.

We used *JDiff* to identify the changes between two consecutive releases of each project, and counted the number of changes that can lead to compilation errors in test cases. Column *Not-Compile* of Table I shows the total amount of changes that may lead to compilation errors in test cases for each project[14]. Column *TCA* indicates the number of changes to which *RepairSignatureChanges* can be applied to repair the compilation errors. *RepairSignatureChanges* can handle a total of 87,338 changes: 33,586 *added parameters*, 18,704 *removed parameters*, 8,430 *parameter type changes* and 26,618 *return type changes*.

On average, for each project, *TCA* can repair the compilation errors for the 45% of the changes that lead to compilation errors. Outliers, i.e., Santuario and Wicket, show that different projects may present a different number of changes in parameter declarations. In 6 of the 9 projects that we considered, more than 40% of the changes that lead to compilation errors are changes in parameter declarations, thus highlighting the usefulness of *TCA*.

[13]Apache Software Foundation, www.apache.org

[14]We do not report the test cases that raise compilation errors because test cases are not available for all the changed methods.

Effectiveness: We applied *TCA* to repair 138 test cases of three different software: JFreeChart, PMD and JodaTime. We considered 6 releases and 21 test cases for JFreeChart, 2 releases and 18 test cases for JodaTime, 13 releases and 99 test cases for PMD. For each release, we executed *TCA* on all test cases that do not compile. The test cases considered in our study were broken by different type of changes: parameter type changes (26), parameter additions (68), parameter removals (23), and return type changes (21).

Change	TC	Errors fixed	Valid test cases	Same as Developers'
Par. Type	26	26	22 (84.62%)	14 (53.85%)
Par. Add	68	68	68 (100.00%)	59 (86.76%)
Par. Remove	23	23	17 (73.91%)	21 (91.03%)
Ret. Type	21	21	21 (100.00%)	11 (52.38%)
Total	138	138	128 (92.75%)	105 (76.08%)

Table II
EFFECTIVENESS ON GENERATING REPAIRS

To measure the effectiveness of *RepairSignatureChanges*, we compiled and ran the test cases repaired by *TCA*, and checked how many test cases compile and execute correctly, i.e., do not fail after repair. We also manually compared the test cases repaired by *RepairSignatureChanges* with the ones repaired by software developers to check if they present the same behavior. Table II shows that all test cases repaired by *RepairSignatureChanges* compile correctly (column *Errors-fixed*), and that 128 test cases (92.75%) also execute correctly (column *Valid-test-cases*). Only 10 of the repaired test cases do not execute. Their non-executability depends on the default values generated by *RepairSignatureChanges* that alter the test behavior, thus causing failures in assertions (4 cases), or runtime exceptions due to the absence of proper initialization of other variables (6 cases). Of the 128 repaired test cases, 105 present the same behavior as the test cases generated by software developers, while the others present a different but valid behavior. All the test cases repaired by *RepairSignatureChanges* cover valid behaviors that integrate the ones covered by the test cases repaired by software developers.

B. Generating test cases for new classes

Applicability: To evaluate the applicability of *TestClassHierarchies* for generating test cases, we analyzed the source code of 5 open source projects. Table III shows the results. Column *Subject* indicates the software version that we investigated, Column *LOC* indicates the size (lines of code) of that version, Column *Classes* indicates the total amount of classes in that version, Column *Hierarchy* indicates the number and percentage of classes that extend a class hierarchy, i.e., classes for which *TestClassHierarchies* can generate test cases.

TestClassHierarchies can generate test cases for 60% of the classes of each project, on average. These results represent an upper bound to the applicability of the approach, in fact the applicability of *TestClassHierarchies* depends not only on the presence of class hierarchies but also on the availability of test cases for classes of a same hierarchy. For example, in JodaTime 1.62, *TestClassHierarchies* could generate test cases for 74% of the classes, but JodaTime does not have test cases for all the implemented classes, thus *TestClassHierarchies* can automatically test 32% of classes.

Subject	LOC	Classes	Hierarchy	
Barbecue 1.5	8,842	55	28	(50.91%)
JFreeChart 1.013	217,357	471	275	(58.39%)
JodaTime 1.62	63,922	99	74	(74.75%)
PMD 4.2	65,279	483	315	(65.22%)
Xstream 1.31	24,655	218	120	(55.05%)

Table III
APPLICABILITY ON TEST GENERATION FOR CLASS HIERARCHIES

Effectiveness: We evaluated the effectiveness of *TestClassHierarchies* by generating test cases for all the classes that belong to a class hierarchy in the five open source systems. For each class considered in the experiment, we removed the test cases implemented by developers for that class, and applied *TestClassHierarchies* to generate new test cases. We compared the test cases generated with *TestClassHierarchies* with the test cases generated by developers, and the test cases generated by three test case generation techniques implemented by tools available online: Randoop version 1.3.2, CodePro version 7.1.0, and EvoSuite version 20110929. Table IV shows the results. Column *C* indicates the number of classes for which we generate the test cases. The other columns indicate the instruction coverage for the class under test obtained with the test cases generated with the different approaches.

The test cases produced by developers achieve the highest coverage, but at a price of a high effort. *TCA* performs as well as EvoSuite, and outperforms both CodePro and Randoop. To characterize the differences between *TCA* and EvoSuite, we compared the set of statements covered by tests cases generated by developers, *TCA*, and EvoSuite for each class under test. The test cases generated by *TCA* and Evosuite cover a common set of 12055 lines of code (49% of the total). *TCA* and EvoSuite are complementary: *TCA* covers 6370 lines (25.9%) not covered by Evosuite, while Evosuite covers 6208 lines not covered by *TCA* (25.2%). *TCA* outperforms Evosuite in 130 test cases, Evosuite outperforms TCA in 134. Evosuite works better than *TCA* when path conditions cannot be covered by copying data used in ancestor test cases, while *TCA* produces test cases that check the software behavior in presence of runtime errors that can be covered only with some domain knowledge that *TCA* implicitly imports from existing test cases.

Subjects	C	**TcA**	Devs	Rand	CodePro	EvoS
Barbecue 1.5	12	**73.42**	63.44	45.80	23.10	82.03
Jfrechart 1.013	204	**47.92**	49.22	27.70	52.06	51.46
JodaTime 1.62	32	**75.28**	88.07	48.97	77.33	71.52
PMD 4.2	56	**47.18**	65.53	40.56	49.68	60.03
Xstream 1.31	36	**58.55**	78.19	18.67	54.18	43.35
Total	340	**60.47**	68.89	36.34	51.27	61.68

Table IV
EFFECTIVENESS OF TEST CASE GENERATION FOR CLASS HIERARCHIES

The test cases produced by developers often stress both the code of the class under test and the code of parent classes, to identify integration faults between parent and child classes. Table IV shows the coverage for the class under test only, but we manually inspected the generated test cases and found that by reusing existing test cases, *TCA* generates test cases that check both the class under test and its integration with parents. Evosuite generate test cases for single classes and produces good test cases for the class, but ignores the integration with the parent class. For example, for the class EthiopicChronology, *TCA* covers 1083 instructions belonging to class EthiopicChronology or one of its parents, while Evosuite covers only 683 instructions.

By modifying developers test cases, *TCA* generates test cases that are more readable than the ones produced by Evosuite. *TCA* test cases are as readable and maintainable as the original tests written by developers. Figure 3 shows a test case generated by *TCA* that although covering a complex behavior is easy to understand thanks to the presence of meaningful names. Figure 6 shows a test case generated by Evosuite that uses abstract names that make test cases difficult to understand and maintain.

```
EthiopicChronology var0 = EthiopicChronology.getInstanceUTC();
assertNotNull(var0);
DateTimeField var1 = (DateTimeField)var0.weekyearOfCentury();
long var2 = var1.addWrapField(-803L, 65533);
assertEquals(var2, 1041465599197L);
```

Figure 6. A test case for class `EthipicChronology` by Evosuite.

VII. CONCLUSIONS

In this paper we introduce the problem of test case evolution that we define as the problem of repairing existing test cases and generating new ones to react to incremental changes in software systems. We identify frequent actions for adapting test cases that developers commonly adopt to repair and generate test cases, and we define five algorithms for evolving test cases as a solution to support software developers. We describe a prototype implementation that we used to validate the approach.

The experimental results presented in the paper show that the approach can repair and generate many test cases, thus reducing the testing effort. *TCA* properly repairs 90% of the compilation errors it addresses, and generates test cases that cover the same amount of instructions of state of the art techniques. The test cases produced by *TCA* are complementary to the ones generated by other techniques, which indicates that *TCA* could be integrated with other approaches to improve testing results.

We experimented the approach with two of the five algorithms. We are extending the prototype to experiment with the complete set of algorithms. We are also mining software repositories to identify other recurrent actions and enrich the set of algorithms.

References

[1] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *Automated Software Engineering*, 2011, pp. 23 –32.

[2] M. Harrold and A. Orso, "Retesting software during development and maintenance," *Frontiers of Software Maintenance*, pp. 99–108, 2008.

[3] A. Pretschner, "Model-based testing," in *International Conference on Software Engineering*, 2005, pp. 722–723.

[4] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *International Symposium on Software Testing and Analysis*, 2008, pp. 15–26.

[5] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion of Conference on Object-Oriented Programming Systems and Applications*, 2007, pp. 815–816.

[6] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "MSeqGen: object-oriented unit-test generation via mining source code," in *Symposium on Foundations of Software Engineering*, 2009, pp. 193–202.

[7] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *International Conference on Fundamental Approaches to Software Engineering*, 2009, pp. 171–185.

[8] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *International Symposium on Software Testing and Analysis*, 2010, pp. 207–218.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[10] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Symposium on the Foundations of Software Engineering*, 2011, pp. 416–419.

[11] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[12] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Transactions Software Engineering Methodology*, vol. 18, no. 2, pp. 1–36, 2008.

[13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th Int. Symp. on Foundations of Software Engineering*. ACM, 2008, pp. 226–237.

[14] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the 18th Int. Symp. on Foundations of Software Engineering*. ACM, 2010, pp. 257–266.

[15] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported an eclipse case study," in *ICSM 2006: Proceedings of IEEE International Conference on Software Maintenance*, 2006, pp. 458–468.

[16] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: refactoring-aware binary adaptation of evolving libraries," in *International Conference on Software Engineering*, 2008, pp. 441–450.

[17] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. ACM, 2008, pp. 481–490.

[18] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. ACM, 2008, pp. 471–480.

[19] M. Mirzaaghaei, F. Pastore, and M. Pezzé, "Algorithms for repairing test suites through parameters adaptation," University of Lugano, Tech. Rep., 2011. [Online]. Available: http://www.inf.usi.ch/phd/mirzaaghaei/TestrepairTR-2011.pdf

[20] G. Denaro, A. Gorla, and M. Pezzè, "DaTeC: Contextual data flow testing of Java classes," in *Companion of International Conference on Software Engineering*, 2009, pp. 421–422.

[21] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," in *International Symposium on Software Testing and Analysis*, 1998, pp. 11–20.

[22] V. Levenshtein, "Binary codes capable of correcting spurious insertions and deletions of ones." *Probl. Inf. Transmission*, vol. 1, pp. 8–17, 1965.

[23] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970.

[24] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatically repairing test cases for evolving method declarations," in *International Conference on Software Maintenance*, 2010, pp. 1–5.

[25] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *International Conference on Automated software engineering*, 2009, pp. 433–444.