Automatically Repairing Test Cases for Evolving Method Declarations

Mehdi Mirzaaghaei[†], Fabrizio Pastore[†], Mauro Pezzè[†‡]

[†]University of Lugano, Lugano, Switzerland
[‡]University of Milano Bicocca, Milano, Italy
Email: {mehdi.mirzaaghaei,fabrizio.pastore,mauro.pezze}@usi.ch

***Abstract*—When software systems evolve, for example due to fault fixes, modification of functionalities or refactoring activities, test cases may become obsolete thus generating wrong results or even not being executable or compilable. Maintaining test cases is expensive and time consuming, and often test cases are discarded by software developers due to high maintenance costs.**

This paper presents *TestCareAssistant*, a technique that combines data-flow analysis with program diffing for automatically repairing test cases that become obsolete because of changes in method declarations (addition, removal, or type modification of parameters or return values). The paper illustrates the efficacy of *TestCareAssistant* by analyzing the impact of method declarations changes on the executability of test cases, and by presenting the preliminary results of applying *TestCareAssistant* to repair 22 test cases.

I. Introduction

Software systems undergo several changes in their lifetime. After the modifications, software engineers re-execute the test cases to identify regressions in the existing functionalities [9]. Unfortunately test cases become obsolete and quickly unusable if not maintained along with the system [1]. Common examples are test cases that do not compile because of changes in method signatures, test scripts that do not execute due to modifications of component APIs, or test cases that do not reveal failures after changes in the specifications.

In the last years several researchers have studied the problem of automatically identifying and correcting obsolete test cases. Some groups focused on GUI test cases. Xie et al. developed some techniques to automatically identify and correct graphical objects affected by GUI changes [20], [8]. Jiang et al. and Memon addressed generic problems like correcting sequences of test scripts, for instance, pressing buttons or invoking methods [15], [12]. Such techniques require accurate system specifications or oracles. Daniel et al. focused on automatically repairing test oracles [3], [2]. These techniques solve important problems related to test maintenance activities that span from repairing GUI test cases, test scripts, and oracles.

In this paper, we deal with the problem of test evolution with particular emphasis on test case compilation errors that depend on changes in parameters or return values, for example due to refactoring of the target system. Some changes of these kind are automatically managed by refactoring techniques [16]. Existing refactoring techniques can prevent simple errors by automating some of the possible refactoring activities like moving or renaming methods, modifying class hierarchies [18], or improving concurrency [5] or reentrancy [19]. Common refactoring practices like adding new parameters to methods [21] are only partially automated by existing tools and techniques. For example, ReBa [6] and Eclipse[1] avoid compilation errors caused by parameter changes only when the modified parameters can be replaced by default values. Empirical studies indicate that 93% of the changes of parameters and return values are performed manually by software developers [17], and this limits the practical applicability of existing techniques. Although current techniques can cope with simple cases, the problem of automatically repairing test case compilation errors is still open.

This paper presents *TestCareAssistant* (*TcA*), a technique that automatically repairs test compilation errors caused by changes in the declaration of method parameters and return values, focusing in particular on insertion and removal and type changes. *TcA* automatically modifies test cases and adapts the parameters according to the modified methods to repair the test case and preserve its behavior. For example if a new parameter is added to a method used in a test case, *TcA* automatically creates and initializes the objects to be passed to the method.

TcA applies both to *pure refactoring* that involves only changes that do not alter software behavior, and *floss refactoring* that combines refactoring and functional changes, which is a software development practice more frequent than *pure refactoring* [17]. *TcA* assumes only that the original functionality is preserved for the given test inputs.

TcA deals with changes that cause compilation errors, thus ignores changes like the addition of a return value, and repairs errors that are not currently fixed by existing refactoring tools, which focus only on simple compilation problems like the ones caused by method renaming or changes in the order of parameters.

This paper contributes to the state-of-the-art by:

- introducing *TestCareAssistant* a technique to repair compilation errors caused by changes in the declaration of parameters and return values;
- analyzing 262 versions of 22 open source systems to

[1]www.eclipse.org

investigate the impact of automated test repair on test maintenance costs;

- evaluating the effectiveness of *TestCareAssistant* by analyzing the preliminary results obtained by repairing 22 test cases.

The paper is structured as follow. Section II introduces *TestCareAssistant*. Sections III to VI detail the technique. Section VII discusses the preliminary empirical results obtained with an early prototype. Section VIII summarizes the main contributions of the paper and the ongoing research work.

II. Repairing Test Cases Automatically

In this section we introduce *TcA* through the simple example of a bank account management system.

```
1  public class BankAccount {
2    private int balance;
3    public void deposit(int cents, String currency){
4      balance += cents * getChange(currency);
5    }
6    public int getBalance(){
7      return balance;
8  }}
```

Listing 1. A simple class that manages a bank account

```
3  public void deposit(Money money, String currency){
4    balance += money.centsValue * getChange(currency);
5  }
```

Listing 2. Changes in class `BankAccount`

```
1  BankAccount account = new BankAccount();
2  int amount = 500;
3  account.deposit(amount,"EUR");
4  assertEquals( 500, account.getBalance() );
```

Listing 3. Test for method `deposit` of class BankAccount in Listing 1

```
2  Money amount = new Money(500);
3  account.deposit(amount,"EUR");
```

Listing 4. Repairing the test case in Listing 3 to adapt to the change

Listing 1 and 2 show a change of parameter type: The type of the parameter of the method `deposit` changes from `int` to `Money`. This change causes a compilation error in the test shown in Listing 3 with the new code, since the test invokes method `deposit` with an integer parameter instead of using the new type (line 3). To verify the modified version of method `deposit`, developers can repair the test in Listing 3 by changing the parameter of method `deposit`. Listing 4 shows the repaired test that passes a `Money` object to method `deposit`.

Although test maintenance activities like the one illustrated above for the BankAccount example are straightforward, the number of test cases to be repaired can be very large and grows while the system evolves. The overall cost of changes grows, and developers often do not have time to update all the test cases along with the modified classes, thus many test cases become obsolete. While software systems evolve, many obsolete test cases diverge from the system and become hard to repair, thus developers need to rewrite new test cases from scratch [1].

TcA combines source code diffing and data flow analysis to repair test cases that do not compile due to changes in parameters or return values. *TcA* works in three steps: it identifies the variable that has to be changed to repair the test case by means of source diffing, determines the correct value to initialize the variable in the new context by means of data flow analysis, and repairs the test case.

We implemented *TcA* as a Java prototype that automatically produces possible fixes for test cases in the form of suggestions for the developers. For example, Listing 5 shows the fix that *TcA* proposes to repair the BankAccount test case. The next sections describe the three steps in details.

```
1  replace:
2  account.deposit(amount,"EUR");
3  with:
4  Money money = new Money(500);
5  account.deposit(money,"EUR");
```

Listing 5. Repair for test in Listing 3

III. Identifying the variables to initialize

When a test case does not compile due to changes in parameters or return values, *TcA* tries to identify the parameters or the return values responsible for the compilation error by simply diffing the original and modified versions of the code. The current *TcA* prototype retrieves the source code of the original and the modified software component through the *svn*[2] version control system and compares them with *JDiff*[3].

Determining the proper values to initialize the modified parameters may be hard or even impossible without additional information, depending on the complexity of the types and the scope of the changes. In particular, parameters of type class can be complex to initialize, because of the presence of many attributes whose initialization values may be difficult to determine. On the other hand, not all the elements of the class may need to be initialized to execute a given test case. For this reason, once identified the modified parameters, *TcA* tries to identify the fields that need to be initialized in order to execute the test cases.

To identify the *program elements to initialize*, *TcA* locates the *first use* of the modified parameter in the new version of the software system. If the modified parameter is an object, *TcA* checks if the first use involves the whole object or one of its fields, to determine the scope of the initialization required to fix the test. If the first use does not involve the whole object, then *TcA* identifies the object attributes used during the execution of the method, and locates their first use. *TcA* uses *Datec* [4] to identify all the uses of the attributes, and then traverses the interprocedural control flow graph [10] provided by *Soot*[4], starting from the modified method, to identify the first use of each attribute.

If the modified parameter is of a primitive type or is an object whose fields are not accessed in the first use, *TcA* identifies the parameter itself as the only program element to initialize.

In the BankAccount example, the parameter `money` is used to access the field `centsValue`. Thus, *TcA* locates all the

[2]http://www.subversion.org
[3]http://www.jdiff.org/
[4]http://www.sable.mcgill.ca/soot/

uses of the fields of the class `Money` that are reachable from the invocation of method `deposit`. In this example only field `centsValue` is accessed within method `deposit`, in line 4 of Listing 2, and thus *TcA* needs to initialize only this attribute.

IV. Identifying Initialization Values

Once identified the *program elements to initialize*, *TcA* determines the proper initialization values –values that preserve the test behavior– by looking for the corresponding values used in the test cases for the original software.

For each *first use* of the *program elements to initialize* identified in the previous step, *TcA* looks for a *corresponding line* in the original software. *TcA* 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, a term, a literal or a variable that replaces the *program element to initialize* in the original software (we call this the *corresponding term*). If the *Unix diff* identifies multiple counterparts for line $L1$, *TcA* selects the line most similar to $L1$ according to the Levenshtein distance [14].

In the BankAccount example, line 4 of Listing 1 is the line that *corresponds* to the use of the field `money.centsValue` at line 4 in Listing 2, and `money.centsValue` is the *term* that corresponds to the integer variable `cents`.

We can determine the value of the *corresponding term* in the original software either statically or dynamically. The *TcA* prototype implements a static approach: It traverses the interprocedural def-use chain [11] of the *corresponding term* until it reaches a definition within the test or within a static initializer, where a constant or a literal is assigned to the defined term. The definition found by traversing the def-use chain matches the value of the *corresponding term* only if the def-use chain represents a mere passing of values from a variable to another (empirical results show that this happens for most of the test cases). If one instruction in the chain performs a computation we cannot determine the value of the *corresponding term* statically. In this case, we can determine the value dynamically, for example by executing the test case on the original software, and tracing its runtime value through a debugger.

In the BankAccount example the inspection of the def-use chain of `money.centsValue` can determine that its value during the original test execution is 500.

If *TcA* does not find a *corresponding line*, it tries to identify the *program elements to initialize* in the modified software starting from the variable defined in line $L1$. If *TcA* does not find a corresponding line for a *program element to initialize* it indicates that the initialization value for this element is *unknown*.

V. Repairing Test Cases

TcA tries to repair a test compilation error by changing the invocation of the modified method to match the new method signature. If the modification consists of adding or changing a parameter, *TcA* adds an instruction to define a new parameter. If the modification consists of adding, changing, or removing a parameter, *TcA* assigns a value to the *program elements to initialize*. If the modification consists in changing the return type, *TcA* retrieves data from test objects instead of using the original return value.

When the new parameter is of primitive type, *TcA* defines a new attribute by simply replacing the original attribute with the new value. When the parameter is an object, *TcA* adds an invocation to the constructor of that object, and automatically assigns values to constructor parameters if they define the *program elements to initialize* identified in the previous steps. To enable the compilation of the test case, *TcA* initializes the constructor parameters that are not used within the test to default values. Finally *TcA* assigns values to the *program elements to initialize* that are not defined by the constructor by invoking proper setter methods, or by using reflection [7] if no proper setter methods are available.

VI. Dealing with Removal of Parameters and Modification of Return Values

The approach described in the previous sections identifies the program elements to be initialized by analyzing the modified software. This is possible when dealing with changes in the type of the parameters or additions of parameters, but not when dealing with removal of parameters or changes of the return type. This Section illustrates how *TcA* deals with these cases.

A typical case of parameter removal occurs when the code refactoring moves a parameter to a class field. Listing 6 shows an example where parameter `currency` of method `deposit` (Listing 2) has been moved to class `Money`. To preserve the test behavior software developers must initialize the new class field with the same value of the removed parameter in the original test, like in Listing 7 where the "`EUR`" value is passed to the `Money` constructor.

Since the modified software does not contain the parameter anymore, we cannot determine the fields to be initialized only from the modified software. *TcA* determines the fields to be initialized following the same steps introduced in Sections III and IV but it starts the analysis from the original code. It first identifies a set of *initialized elements* by locating the use of either the removed parameter or of its fields in the original software as described in Section III. In the running example, the parameter `currency` is used at line 4 of Listing 2. Then for each *initialized element*, *TcA* finds the *corresponding term* in the modified software, i.e. the element that replaces the original parameter or one of its fields in the modified version: `money.currency` in Listing 6 at line 4. *TcA* inspects the interprocedural def-use chain of the *corresponding term* to identify the methods that set its value, and to determine the value to use to initialize the new element.

```
3  public void deposit(Money money){
4    balance +=money.centsValue*getChange(money.currency);
5  }
```

Listing 6. Parameter removal for method `deposit` of Listing 2

```
BankAccount account = new BankAccount();
Money amount = new Money(500,"EUR");
account.deposit(amount);
assertEquals( 500, account.getBalance() );
```

Listing 7. Test for the modified method `BankAccount.deposit`

```
public class BankAccount {
  private int balance;
  public void deposit(int money, String currency){
    balance += money * getChange(currency);
  }
  public Money getBalance() {
    Money m = new Money( balance );
    return m;
}}
```

Listing 8. Return type change for method `BankAccount.deposit`

```
public static class Money{
  public final int centsValue;
  public Money(int amountInCents){
    this.centsValue=amountInCents;
}}
```

Listing 9. The `Money` type returned by the modified method `deposit`

Listing 8 shows an example of return type change: The return type for method `getBalance` of Listing 1 has been changed from `int` to `Money`. The change causes a compilation error at line 4 of the test case in Listing 3. The example shows that while parameter changes can affect the uses in the modified method, changes in the return type can affect the uses in the original test cases, and thus *TcA* applies the repair process starting from the original test case.

TcA first inspects the original test case to determine the *used elements*, that is the return value or some of its fields, and then inspects the def-use chain of the *used elements* to identify their definition within the original method. In the *BankAccount* example *TcA* detects that the return value is used at line 4 of the test case in Listing 3, and that it is defined in two statements of Listing 1: at lines 2 and 4.

For each definition of the *used element* in the original software, *TcA* identifies a *corresponding term* in the modified software. Then, *TcA* traverses the interprocedural def-use chain of the *corresponding term* to determine how to retrieve its runtime value within the test. For each definition in the chain, *TcA* checks if it defines a field of an object accessible within the test case. Then *TcA* identifies the instruction to be used to access its runtime value to determine the fix.

In the BankAccount example, *TcA* detects that field `balance` in Listing 8 is the *corresponding term*, and it is used to define the field `centsValue` in the `Money` constructor during the invocation of the method `getBalance` from the test case. The field `centsValue` can be accessed from the test because a `Money` object is returned by method `getBalance`. Then, *TcA* determines that the field `Money.centsValue` can be directly accessed from the test case, and builds the repair shown in Listing 10.

```
replace:
assertEquals(500, account.getBalance());
with:
assertEquals(500, account.getBalance().centsValue);
```

Listing 10. Repair for return type change.

VII. Empirical Evaluation

To evaluate the approach proposed in this paper, we must answer two main research questions: "What is the impact of changes in the declaration of parameter and return values on software evolution?" and "To what extent our approach can fix test cases automatically?"

To answer the first question, we analyzed 262 versions of 22 open source projects of the Apache Software Foundation[5]. We used *JDiff* to automatically extract the changes between two consecutive releases of each project. *JDiff* identifies the following differences: methods added/deleted/renamed, class attributes and code blocks inserted/removed, imports added/removed/renamed, parameters and return values added/removed/renamed. In our experiments, *JDiff* identified 95,067 differences, 7.7% of which regarded parameter and return value declarations: 2,376 parameters added to methods, 2,284 parameters removed, 1,464 parameter types changed, 1,218 return types changed. These data indicate that declarations changes are relevant, as confirmed by similar experiments conducted on six versions of Eclipse by Xing et al. [21].

We classified software versions according to the number of parameter changes, and we discovered that changes occur with various frequency in different releases: 53% of the releases do not present any change, 37% present less than 50 parameter/return type changes each (for a total of 1,178, the 16% of all the parameter type changes), and 10% present more than 50 parameter changes each, the 83% of all the changes. These data reflect the different maturity of the software systems and indicate that usually the impact of parameter and return type modifications is limited, but there are several cases in which these modifications are so prominent that the effort required to maintain test suites could heavily impact on software development costs.

To answer the second question, we conducted a preliminary study on 22 test cases of 6 software systems: Continuum[6], Geronimo[7], xml-security[8], PMD[9], POI[10], and Shindig[11].

The considered test cases suffered from 24 compilation errors due to changes in the declaration of 8 methods. In the study we selected methods to cover the type changes that *TcA* should deal with: primitive parameters added, primitive parameters replaced by objects, object parameters added, etc. The test cases failed because of the following changes: 9 parameter types changed, 8 parameters added, 3 parameters removed, 4 return types changed.

We evaluated *TcA* as follows: we executed *TcA* on the test cases; we applied the correction proposed by tool; we compiled and executed the corrected test cases; and we inspected the results. We evaluated the solution automatically generated by *TcA* by checking the validity of the corrections,

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

[6]http://continuum.apache.org/

[7]http://geronimo.apache.org/

[8]http://santuario.apache.org/

[9]http://pmd.sourceforge.net/

[10]http://poi.apache.org/

[11]http://shindig.apache.org/

and we evaluated the correct initialization of the variables by comparing the *TcA* suggestions with the changes applied by the application developers.

The fixes that *TcA* produced automatically solved 18 out of the 24 compilation errors, corresponding to 16 out of the 22 test cases (72% of the total). By manually inspecting the fixed test cases we noticed that all the generated fixes produce correct test cases. These preliminary results suggest that *TcA* presents an acceptable success rate and does not lead to false positives.

By comparing the initializations suggested by *TcA* with the actual corrections of the application developers, we noticed that *TcA* correctly initializes 29 out of the 36 variables (80%) required to correctly fix the test cases: 14 variables were initialized by replacing new parameters with the default ones, and 15 by automatically deriving the values from the original test. *TcA* was not able to repair 6 test cases, because the parameter values cannot be determined by traversing def-use chains. Two test cases could not be repaired because they refer to methods whose logic was completely changed in the modified version of the code. One of them, for example, calls a method in which a parameter of type `String` containing an ID is replaced with a `File` object that should point to a file containing the ID. Without any knowledge about the format of the data stored in the file, *TcA* cannot automatically repair the test case. In two cases `Map` objects are used to retrieve or store modified parameters and return values. Also in these cases, *TcA* would need specific knowledge to determine the key needed to retrieve the values from the maps and repair the test cases. The last two unsuccessful cases depend on current limitations of the *TcA* prototype. In one case the modified objects are inserted in a collection, which is not analyzed by the data flow analyzer embedded in *TcA*. In the other case the modification affects an interface definition, currently not managed by *TcA*.

VIII. Conclusions

This paper proposes *TestCareAssistant*, a technique that automatically repairs test case compilation errors caused by changes in method declarations.

The preliminary results presented in this paper indicate that the approach can effectively solve some classes of compilation problems, and highlight some limitations that comprise our research plan. Data flow analysis deals well with simple data structures, but is less effective for complex data structures, external components, and devices like hash tables, databases, or files. We are currently working on the combination of data-flow analysis with dynamic analysis to overcome these limitations.

We consider *TestCareAssistant* only the first step towards a holistic approach for test suites repair and evolution. We are currently working on extending *TestCareAssistant* to automatically augment the test cases for methods with signature changes, because they tend to be more fault prone than methods without signature changes [13].

References

[1] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 571–579. ACM, 2005.

[2] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA '10: International Symposium on Software Testing and Analysis*, 2010.

[3] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *ASE'09: Proceedings of the 24th IEEE/ACM international Conference on Automated Software Engineering*. IEEE/ACM, 2009.

[4] G. Denaro, A. Gorla, and M. Pezzè. Datec: Contextual data flow testing of java classes. In *Companion of the Proceedings of 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[5] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 397–407. IEEE Computer Society, 2009.

[6] D. Dig, S. Negara, V. Mohindra, and R. Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 441–450. ACM, 2008.

[7] I. R. Forman and N. Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[8] C. Fu, M. Grechanik, and Q. Xie. Inferring types of references to gui objects in test scripts. In *ICST '09: International Conference on Software Testing Verification and Validation*, pages 1–10. IEEE Computer Society, 2009.

[9] M. Harrold and A. Orso. Retesting software during development and maintenance. *FoSM 2008: Frontiers of Software Maintenance 2008*, pages 99–108, 2008.

[10] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 11–20. ACM, 1998.

[11] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, 1994.

[12] B. Jiang, T. H. Tse, W. Grieskamp, N. Kicillof, Y. Cao, and X. Li. Regression testing process improvement for specification evolution of real-world protocol software. In *QSIC 2010: Proceedings of the 10th International Conference on Quality Software*. IEEE Comp. Soc., 2010.

[13] S. Kim, E. J. Whitehead, and . J. Bevan, Jr. Properties of signature change patterns. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 4–13. IEEE Computer Society, 2006.

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

[15] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions Software Engineering Methodology*, 18(2):1–36, 2008.

[16] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 2004.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 287–297. IEEE Computer Society, 2009.

[18] M. Streckenbach and G. Snelting. Refactoring class hierarchies with kaba. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 315–330. ACM, 2004.

[19] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of soft. eng.*, pages 173–182. ACM, 2009.

[20] Q. Xie, M. Grechanik, and C. Fu. Rest: A tool for reducing effort in script-based testing. In *ICSM'08: IEEE International Conference on Software Maintenance*, pages 468–469. IEEE Computer Society, 2008.

[21] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported an eclipse case study. In *ICSM'06: IEEE International Conference on Software Maintenance*, pages 458–468. IEEE Computer Society, 2006.