Multi-Level Mutation Testing of Java and AspectJ Programs Supported by the Proteum/AJv2 Tool

Filipe Gomes Leme^1, Fabiano Cutigi Ferrari^2
José Carlos Maldonado^1, Awais Rashid^3

^{1}Computer Systems Department – University of São Paulo (ICMC/USP) – Brazil

^{2}Computing Department – Federal University of São Carlos (UFSCar) – Brazil

^{3}Computing Department – Lancaster University – United Kingdom

leme.fg@gmail.com, fabiano@dc.ufscar.br,
jcmaldon@icmc.usp.br, marash@comp.lancs.ac.uk

Abstract. *The application of testing techniques and the associated test selection criteria strongly relies on adequate tooling support. This paper describes the evolution of Proteum/AJ, a tool originally conceived to support the mutation testing of aspect-oriented (AO) programs. Proteum/AJ automates the application of AspectJ-specific mutation operators. Its evolution, named Proteum/AJv2, also supports the application of unit mutation operators in both Java (object-oriented) and AspectJ programs through a newly graphical user interface (GUI). We show how the tool architecture and the use of design patterns facilitated the addition of mutation operators as well as the GUI development. We also describe results of a preliminary assessment study that comprised the application of both traditional (unit) and AO-specific mutation operators in a complete mutation testing cycle.*

Demo Video: `http://www.youtube.com/watch?v=v1hxRKifXbc`

1. Introduction

Historically, mutation testing [5] has shown to be an effective test selection criterion to evaluate existing test sets [10]. Furthermore, it also leads to the creation of test sets that are effective to reveal software faults. Using mutation operators, testers can create various slightly modified versions of a program, called mutants. Each mutant contains a single fault and composes the set of test requirements that should be covered by the test set. Note that due to the large number of mutants that testers must handle (*i.e.* create, execute and analyse), this testing criterion strongly relies on automated mechanisms.

In previous research [6], mutation testing was customised to software developed with aspect-oriented programming (AOP), a relatively recent software development technique [11]. The *Proteum/AJ* tool [7] was built to support the application of the approach to programs written in AspectJ, which represents a mainstream AOP technology. The tool automates a set of AspectJ-specific mutation operators [6], and supports all steps of mutation testing, such as mutant generation, test case execution and mutant analysis. Note that such operators target basic AO elements such as pointcuts and advices. As a consequence, they impact on the interfaces between aspects and the base code (*i.e.* their communication points), so that testing using such operators can be classified at the integration level.

This paper describes *Proteum/AJv2*, which is an evolution of *Proteum/AJ* to support the mutation testing of both object-oriented (OO) and aspect-oriented (AO) programs. The novel features of *Proteum/AJv2*, when compared to its predecessor, are: (i) a set of unit mutation operators that can be applied to both Java and AspectJ programs; (ii) a newly developed graphical user interface; (iii) enhanced visualisation and manipulation of mutation targets and mutants; (iv) customised test results reports; and (v) simultaneous management of multiple test projects. To the best of our knowledge, *Proteum/AJv2* is the only tool that automates the mutation testing of programs developed under these two paradigms. Thus, it represents a step towards the integrated testing of OO and AO programs.

In this paper, Section 2 describes the tool architecture and how it eases the addition of new mutation operators as well as the new GUI development. Section 3 describes the process of creating mutants in *Proteum/AJv2* and how this is supported by the GUI. An evaluation study, together with the obtained results, is presented in Section 4. To conclude, we compare *Proteum/AJv2* with other mutation testing tools (Section 5) and present some final remarks (Section 6).

2. Proteum/AJv2 Architecture

The tool architecture partially implements a reference architecture for software testing tools called *RefTEST* [13]. *RefTEST* is based on the separation of concerns (SoC) principles, on the Model-View-Controller (MVC) and three-tier architectural patterns, and on the ISO/IEC 12207 standard. *Proteum/AJv2* largely benefits from the reuse of domain knowledge contained in *RefTEST* (organisational and supporting modules) and its easy integration with other life-cycle tools (e.g. requirements and implementation), which has guided the tool structuring in terms of functionalities and module interactions.

The *Proteum/AJv2* architecture is depicted in Figure 1. To clarify, *Proteum/AJv2* does not currently include features defined in *RefTEST* related to supporting and organisational activities[1] (e.g. documentation and configuration management). *RefTEST* also requires a set of service tools (e.g. persistence and access control), which is partially implemented with support of the iBATIS framework[2] for data persistence.

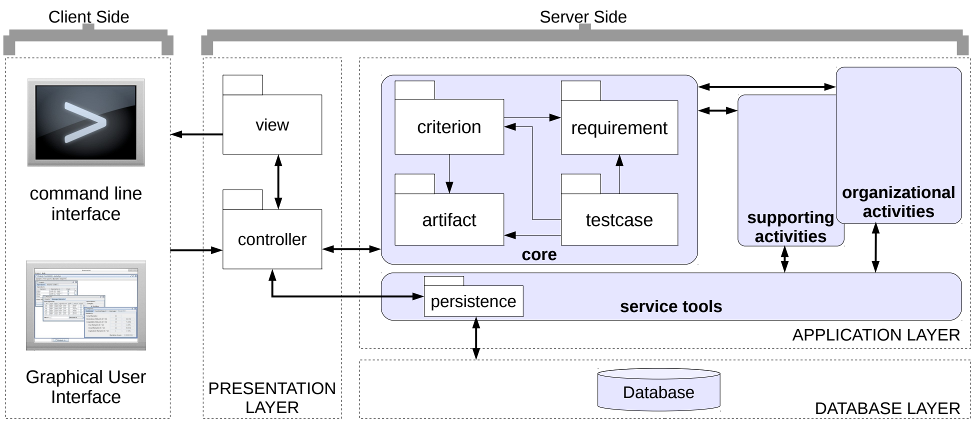


Figure 1: *Proteum/AJv2* Architecture

The *Server Side* includes MVC modules. The alternative user interfaces appears on the *Client Side*. The core comprises the main concepts that should be handled by testing tools, as proposed in *RefTEST*. In *Proteum/AJv2*, *criterion* maps to mutation testing, *artefact* maps to source code, *requirement* maps to mutant, and *test case* maps to the test case itself. For example, the *criterion* module handles the mutant generation, compilation and analysis, while the *test case module* handles test case execution and and evaluation. Changes required to provide support for unit operators and to build the GUI are described next.

The Mutation Engine: Since the tool architecture was designed with special attention to the evolution and maintenance properties, adding support to unit mutation operators has not required drastic changes in the original structure. Figure 2 shows a simplified UML diagram

[1]More details can be found elsewhere [13].

[2]`http://attic.apache.org/projects/ibatis.html` (07/05/2015).

that illustrates some internal details of the *Proteum/AJv2* mutation engine. The classes with grey background represent additions implemented in this version of the tool. Each mutation operator is encapsulated within a specific class. Table 1 lists the unit mutation operators supported by *Proteum/AJv2*. More details about the operators are given in Section 3.

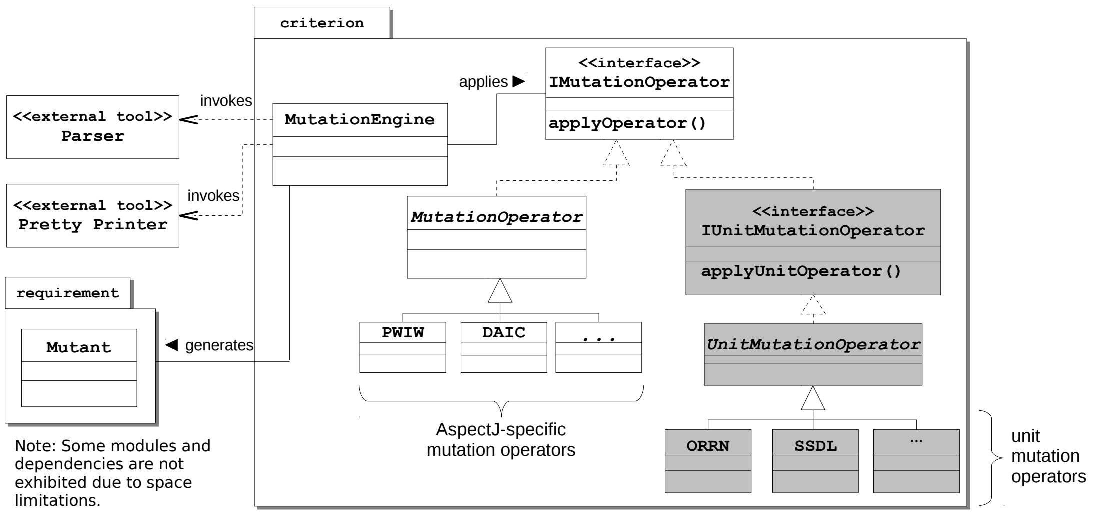


Figure 2: Details of the *Proteum/AJv2*'s mutation engine

The tool allows the selection of specific internal targets (methods and advices) to be mutated. This is realised through the `IUnitMutationOperator` interface, which contains a method whose parameters include a list of internal target names. This enforces client classes to provide such a list. It is implemented by each unit mutation operator, *i.e.* the children of the `UnitMutationOperator` abstract class.

The Graphical User Interface: *Proteum/AJv2* brings a newly developed graphical user interface (GUI). The creation of GUIs helps to increase the adoption of testing tools, which otherwise represents a barrier for both academia and industry [9].

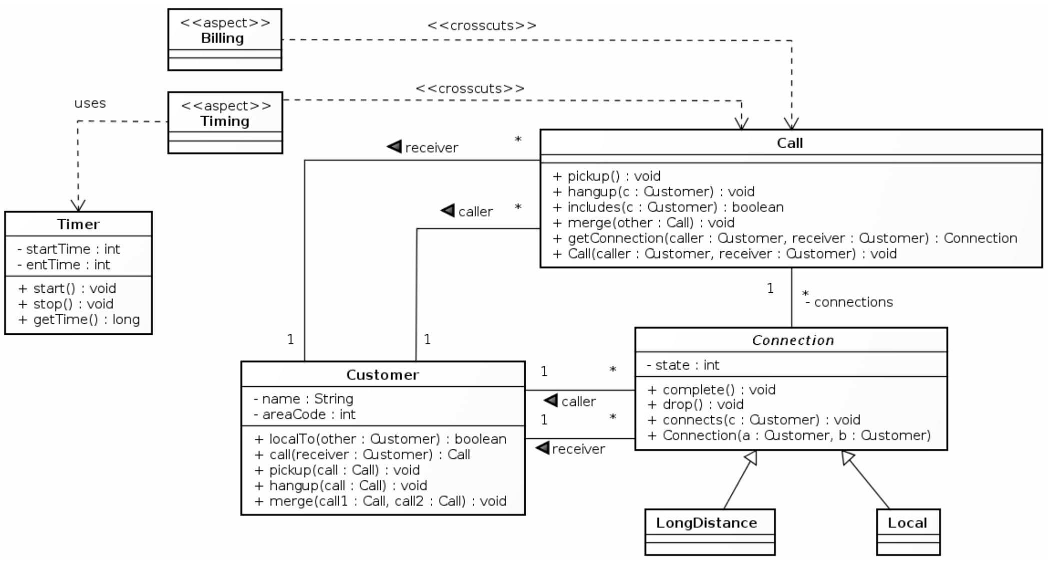
The set of requirements for the *Proteum/AJv2* GUI was derived from existing testing tools such as *Proteum* [4] and *JaBUTi* [16]. These requirements can be divided into: (i) management of test projects; (ii) test case handling (e.g. addition and visualisation); (iii) test requirement generation (mutants); and (iv) report generation. To meet such requirements, we used the *Mediator*, *Observer*, *Command* and *Singleton* design patterns [8]. *Observer* controls the updates of graphical components in the interface. *Command* and *Mediator* encapsulate events related to the graphical components and handle such events. Finally, *Singleton* guarantees the access to the main frame to the other components.

3. Proteum/AJv2 Main Features

This section describes the main features of *Proteum/AJv2* in terms of the mutation process supported by the tool and how this process is enhanced by the addition of the new features listed in Section 1. Note that *Proteum/AJv2* licence is still under definition; the reader can contact the authors for further information.

Running Example: To illustrate the usage of *Proteum/AJv2* and its features, we selected a customised version of the *Telecom* application [2], which is originally distributed together the AspectJ tools. *Telecom* simulates telephony calls and conferences between two or more local and long distance customers.

Figure 3 depicts the classes and aspects included in the application. The main basic classes are `Call`, `Customer` and `Connection` (`Local` or `LongDistance`), which model the basic entities of the system. The `Billing` and `Timing` aspects implement the billing and timing concerns, respectively, with support of `Timer`, a stopwatch functionality.



<<aspect>> Billing
<<crosscuts>>
uses
<<aspect>> Timing
<<crosscuts>>
receiver
Call
+ pickup() : void
+ hangup(c : Customer) : void
+ includes(c : Customer) : boolean
+ merge(other : Call) : void
+ getConnection(caller : Customer, receiver : Customer) : Connection
+ Call(caller : Customer, receiver : Customer) : void
caller
Timer
- startTime : int
- entTime : int
+ start() : void
+ stop() : void
+ getTime() : long
connections
Connection
- state : int
+ complete() : void
+ drop() : void
+ connects(c : Customer) : void
+ Connection(a : Customer, b : Customer)
Customer
- name : String
- areaCode : int
+ localTo(other : Customer) : boolean
+ call(receiver : Customer) : Call
+ pickup(call : Call) : void
+ hangup(call : Call) : void
+ merge(call1 : Call, call2 : Call) : void
caller
receiver
LongDistance
Local

Figure 3: Class diagram of the *Telecom* application

The Mutation Process Supported by Proteum/AJv2: Mutation testing has a high computational cost due to the execution and analysis of numerous mutants. This motivates researchers to spend effort to identify the so-called *sufficient mutation operators* [14], which have low costs (in terms of the number of produced mutants) and high effectiveness in revealing faults [14].

Based on this previous knowledge, the list of unit mutation operators introduced in *Proteum/AJv2* consists of a group of sufficient operators identified for C programs [3, 14] and further validated by Vincenzi [15][3]. They are listed in Table 1[4]. Full descriptions can be found in the original report by Agrawal et al. [1].

Table 1: Unit mutation operators implemented in *Proteum/AJv2*

Operator	Description
CGCR	Constant replacement using global constants
CLCR	Constant replacement using local constants
CGSR	Scalar variable replacement using global constants
CLSR	Scalar variable replacement using local constants
VDTR	Adds a trap function in scalar variables to test if variable will be zero, positive and negative
VTWD	Changes the value of scalar variables for its value predecessor / successor
OASN	Replacement of an arithmetic operator with shift operator
OEBA	Replacement of a plain assignment with bitwise assignment
ORRN	Replacement of a relational operator with other relational operator
SDWD	Statement `do-while` replacement with `while` statement
SMTC	Adds a statement in every loop to force execute at most *n* times
SSDL	Systematically removes each statement

Once the mutants are created, the user can compile the mutants and run *JUnit* test cases, for which the tool will collect the results and compare them to the results from the original application. Note that for some AspectJ-specific mutation operators, the tool is able to automatically detect equivalent mutants through the analysis of join point shadows [7].

[3]According to Vincenzi [15], the OLBN operator [1] – which is included in the sufficient set – is not applicable to Java programs. The OEBA operator, on the other hand, has proved to be efficient in some studies [3], and hence has been included in *Proteum/AJv2*.

[4]The AspectJ-specific mutation operators [6] of *Proteum/AJ* are also available in *Proteum/AJv2*.

The full set of features introduced in *Proteum/AJv2* are not herein described due to space constraints. Amongst them, we highlight: (i) multiple test project creation and management; (ii) selective mutant execution; (iii) individual test case execution; (iv) test case activation/deactivation; (v) mutant visualisation; and (vi) source class/aspect inclusion/exclusion. *Proteum/AJv2* can also be operated through a command line interface, although this paper focuses on the GUI, as next presented.

GUI Features: To create a new test project, the user needs to first create a compressed file (in ZIP format) that includes the full source code of the system under testing (SUT) and also compilation and test execution directives, defined as Apache *Ant*[5] tasks. Once the SUT is submitted to the tool, test project customisation in several ways is facilitated by the GUI (illustrated in Figure 4a).

In the *Targets* window (Figure 4b), the user can select which source files will be tested and also choose the mutation operators to be applied to the SUT. For each source file, the user can select internal elements (methods and advices) to be mutated.

The *Mutants* window exhibits the set of mutants for the current test project. In the *Manage Mutants* tab, the user can check the details of each mutant such as the original and the mutated source files, the differences between these files (Figure 4c), and the mutant status. The user can also filter the mutants by mutation operators. On the *Create* tab, options related to mutant generation (e.g. re-generating all mutants or generating mutants for newly selected mutation operators) are available.

The *Reports* window shows reports regarding the current test project results. For example, it shows the number of generated mutants, the number of anomalous (i.e. non-compilable) mutants, the number of dead mutants and the current mutation score, which is the rate of dead mutants in relation to the total number of non-equivalent, generated mutants. The tool also permits the user to generate an external report with results filtered by mutant status. On the *Coverage* tab, the user can analyse the mutant scores broke down either by applied operators (Figure 4d) or test cases.

Finally, the *Test Cases* window (not shown in Figure 4) allows the selection of test case files to be executed (*Ant* directives are provided with the SUT), inclusion of new test case files and the activation/deactivation of test cases.

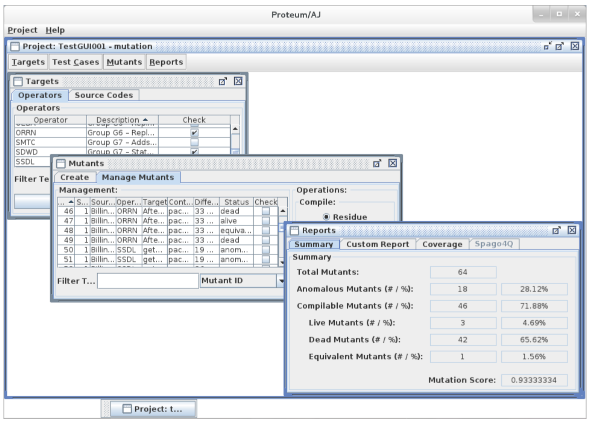
4. Evaluation

This section describes an evaluation study of *Proteum/AJv2*. The study aimed a preliminary assessment of the tool in terms of completeness and correctness. To achieve this goal, we executed a full mutation testing process using *Proteum/AJv2* over two versions of the *Telecom* application, early introduced in Section 3. The first version is purely object-oriented (implemented in Java), while the second includes some concerns modularised with aspects (implemented in AspectJ), as depicted in Figure 3. Note that the OO version of *Telecom* was obtained by inserting aspect-related behaviour into the core classes such as `Call`, `Connection` and `Customer`.

Applied methodology: After setting up the *Telecom* application to be used during the study, we systematically defined functional test cases by analysing and identifying our system inputs and outputs, their valid and invalid equivalence classes and boundary values. Then, we implemented test cases to cover such test conditions using *JUnit* for both projects (*Telecom OO* and *Telecom AO*).

With both applications ready to be tested with *Proteum/AJv2*, we applied the mutation testing considering three different scenarios, as shown in Table 2. Scenarios 1 and 2

[5]`http://ant.apache.org/` (07/05/2015).



Operators

Operator	Description	Check
ORRN	Group G6 – Repl...	✓
SMTC	Group G7 – Adds...	
SDWD	Group G7 – Stat...	✓
SSDL		

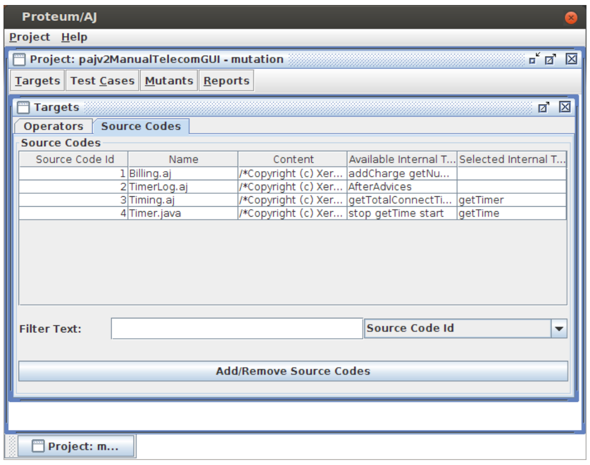
Management:

...	S...	Sour...	Oper...	Target	Cont...	Diffe...	Status	Check
46	1	Billin...	ORRN	Afte...	pac...	33 ...	dead	
47	1	Billin...	ORRN	Afte...	pac...	33 ...	alive	
48	1	Billin...	ORRN	Afte...	pac...	33 ...	equiva...	
49	1	Billin...	ORRN	Afte...	pac...	33 ...	dead	
50	1	Billin...	SSDL	get...	pac...	19 ...	anom...	
51	1	Billin...	SSDL	get...	pac...	19 ...	anom...	

Summary

Total Mutants:	64	
Anomalous Mutants (# / %):	18	28.12%
Compilable Mutants (# / %):	46	71.88%
Live Mutants (# / %):	3	4.69%
Dead Mutants (# / %):	42	65.62%
Equivalent Mutants (# / %):	1	1.56%
Mutation Score:		0.93333334

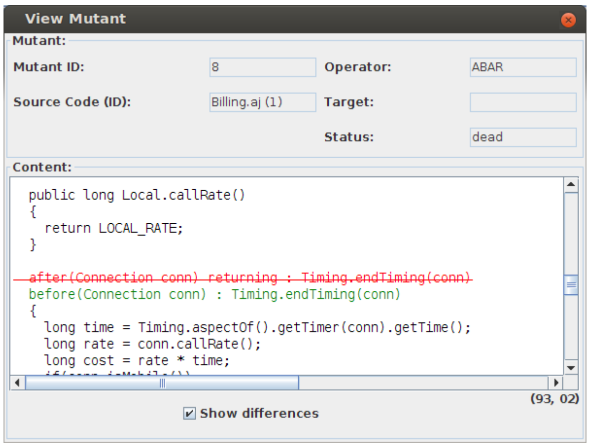
(a) *Proteum/AJv2* GUI overview



Source Code Id	Name	Content	Available Internal T...	Selected Internal T...
1	Billing.aj	/*Copyright (c) Xer...	addCharge getNu...	
2	TimerLog.aj	/*Copyright (c) Xer...	AfterAdvices	
3	Timing.aj	/*Copyright (c) Xer...	getTotalConnectTi...	getTimer
4	Timer.java	/*Copyright (c) Xer...	stop getTime start	getTime

Filter Text: Source Code Id

(b) Source code selection tab



Mutant ID: 8 Operator: ABAR

Source Code (ID): Billing.aj (1) Target:

Status: dead

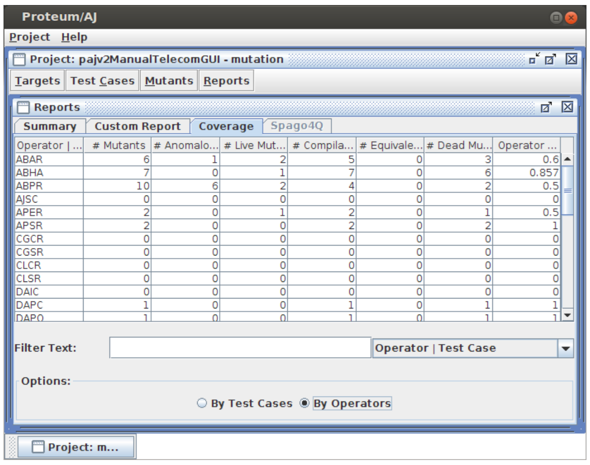
```
public long Local.callRate()
{
  return LOCAL_RATE;
}

~~after(Connection conn) returning : Timing.endTiming(conn)~~
before(Connection conn) : Timing.endTiming(conn)
{
  long time = Timing.aspectOf().getTimer(conn).getTime();
  long rate = conn.callRate();
  long cost = rate * time;
```

(93, 02)

[✓ Show differences]

(c) Mutant difference view


Proteum/AJ
Project Help
Project: pajv2ManualTelecomGUI - mutation
Targets Test Cases Mutants Reports
Reports
Summary Custom Report Coverage Spago4Q
Options:
Project: m...

Operator | ...	# Mutants	# Anomalo...	# Live Mut...	# Compila...	# Equivale...	# Dead Mu...	Operator ...
ABAR	6	1	2	5	0	3	0.6
ABHA	7	0	1	7	0	6	0.857
ABPR	10	6	2	4	0	2	0.5
AJSC	0	0	0	0	0	0	0
APER	2	0	1	2	0	1	0.5
APSR	2	0	0	2	0	2	1
CGCR	0	0	0	0	0	0	0
CGSR	0	0	0	0	0	0	0
CLCR	0	0	0	0	0	0	0
CLSR	0	0	0	0	0	0	0
DAIC	0	0	0	0	0	0	0
DAPC	1	0	0	1	0	1	1
DAPO	1	0	0	1	0	1	1

Filter Text: Operator | Test Case

By Test Cases [✓ By Operators]

(d) Coverage report by mutation operator

Figure 4: Examples of windows available in the *Proteum/AJv2*'s GUI.

address the newly included OO operators and scenario 3 encompass only AO mutation operators In order to gain confidence on the evaluation results, we followed the same procedure twice and, as expected, results for the two rounds were identical.

To execute each of those scenarios we followed six steps: (A) Creation of a test project; (B) Selection of desired targets including classes, aspects, operations (all methods and advices were selected) and mutation operators; (C) Generation and compilation of mutants; (D) Execution of mutants using the functional-adequate test set; (E) Creation of new test cases for live mutants (mutant adequate set); (F) Classification of equivalent mutants.

Results: After applying the planned steps to all three scenarios, we were able to gather and synthesise the results presented in Table 3. The table shows the evolution of the mutant sets during the last four steps. For example, the number of mutants for the second scenario was 251, in which unit operators were applied to classes and aspects of *Telecom AO*. After running the initial test set (step D), 112 mutants were killed and 71 remained alive. The enhancement of the test set (step E) killed more 41 mutants (153 killed, in total). The remaining mutants were set as equivalent (step F), thus reaching a mutation score of 1.0.

The sizes of the new mutation-adequate test sets for each of the projects (step E) are shown in Table 4. For example, in the second scenario (*i.e.* testing of *Telecom AO* using

Table 2: Selected targets for each scenario

Application	Scenario	Targets		
		Classes	Aspects	Operators
Telecom OO	OO testing (1)	Call, Connection, Customer, Local, LongDistance, Timer	-	CGCR CLCR CGSR CLSR VDTR VTWD OASN OEBA ORRN SDWD SMTC SSDL
Telecom AO	OO testing (2)	Call, Connection, Customer, Local, LongDistance, Timer	Billing, Timing	CGCR CLCR CGSR CLSR VDTR VTWD OASN OEBA ORRN SDWD SMTC SSDL
Telecom AO	AO testing (3)	Call, Connection, Customer, Local, LongDistance, Timer	Billing, Timing	ABAR ABHA ABPR AJSC APER APSR DAIC DAPC DAPO DEWC DSSR PCCC PCCE PCCR PCGS PCLO PCTT POAC POEC POPL PSDR PSWR PWAR PWIW

Table 3: Results

Application	Telecom OO				Telecom AO - OO testing				Telecom AO - AO testing			
Execution Step	C	D	E	F	C	D	E	F	C	D	E	F
Anomalous	80	80	80	80	68	68	68	68	6	6	6	6
Alive	218	81	29	0	183	71	30	0	55	17	8	0
Dead	0	137	189	189	0	112	153	153	0	11	20	20
Equivalents	0	0	0	29	0	0	0	30	0	27	27	35
Total	298	298	298	298	251	251	251	251	61	61	61	61
Mutation Score	0	0.628	0.867	1.0	0	0.612	0.836	1.0	0	0.393	0.714	1.0

only unit operators), we added seven new test cases to achieve full mutation score. Note that the final test set was not revised to be an optimal (*i.e.* minimum) set.

Table 4: Number of tests cases on each evaluation step.

Test case set size	***Telecom OO***	***Telecom AO* - Unit**	***Telecom AO* - Aspect**
Initial functional	22	22	22
Mutation adequate	13	7	1
Final set size	**35**	**29**	**23**

This case study allowed us to verify that *Proteum/AJv2* is a tool capable of supporting the entire mutation testing process: from mutant generation, test case execution to mutant analysis. Also, the new GUI proved to be very helpful for quickly customising the testing targets, specific mutant execution and analysis and for generating test reports.

5. Related Work and Limitations

To the best of our knowledge, *Proteum/AJv2* is the first mutation testing tool that supports unit mutation testing of both Java and AspectJ programs, as well as AspectJ-specific mutations. Other tools like *Jester*[6], $\mu Java$[7] and *PIT*[8] only support the mutation testing of Java programs. In particular, $\mu Java$ supports mutation testing at the unit ant class levels, while the others only address the unit level. Besides them, *AjMutator*[9] is a tool that supports the mutation testing of AspectJ programs based on a subset of the AspectJ-specific mutation operators implemented in *Proteum/AJv2*.

[6]`http://jester.sourceforge.net/` (07/05/2015).
[7]`http://cs.gmu.edu/~offutt/mujava/` (07/05/2015).
[8]`http://pitest.org/` (07/05/2015).
[9]`http://www.irisa.fr/triskell/Software/protos/AjMutator/` (07/05/2015).

However, these tools have limitations specially related to the test project management and results storage. *Proteum/AJv2* leverages previous knowledge on mutation tools from its developers' research group, thus allowing for a wide range of test project configurations, as well as experimental procedures through its interfaces.

One negative aspect of *Proteum/AJv2* is that it requires an infrastructure setup to execute. It includes the installation of the *AspectJ-front* toolkit and database setup. Besides, *Proteum/AJv2* can only be executed in Linux-based environments, due to its dependence on *AspectJ-front*. The integration of *Proteum/AJv2* with the Eclipse IDE and the development of a pure Java parser could resolve that issue.

6. Final Remarks

We described the main features of the *Proteum/AJv2* tool that supports the mutation testing of Java and AspectJ programs. Similar to its predecessor [7], *Proteum/AJv2* leverages previous knowledge on the development of testing tools [4, 16] and on reference architectures [13] to configure an integrated environment for testing Java and AspectJ applications.

The presented version of the tool was used to support a master's study conducted by Lacerda that investigated the cost reduction of mutation testing for AO programs [12]. Feedback from Lacerda and Ferrari allowed us to not only fix some bugs but also implement improvements related to the GUI usability.

We are currently planning a series of improvements in *Proteum/AJv2* to support experimentation. This includes creating parameterised scripts to execute a chain of tasks such as test project creation, mutation generation, test case execution and coverage analysis. The design of customised reports are also included in the improvement plan.

Acknowledgements

We thank the financial support received from CNPq (Universal Grant #485235/2013-7), FAPESP (grants #2011/21515-3 and #05/55403-6), and UFSCar (RTN grant).

References

[1] Agrawal et al., H. (1989). Design of mutant operators for the C programming language. Tech. Report SERC-TR41-P, Purdue University, West Lafayette/IN - USA.

[2] Alves, P., Figueiredo, E., and Ferrari, F. C. (2014). Avoiding code pitfalls in aspect-oriented programming. In *SBLP'14*, pages 31–46 (LNCS v.8771). Springer.

[3] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verif. & Reliab.*, 11(2):113–136.

[4] Delamaro, M. E. and Maldonado, J. C. (1996). Proteum: A tool for the assessment of test adequacy for C programs. In *PCS Conference*, pages 79–95.

[5] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43.

[6] Ferrari, F. C., Maldonado, J. C., and Rashid, A. (2008). Mutation testing for aspect-oriented programs. In *ICST'08*, pages 52–61. IEEE.

[7] Ferrari, F. C., Nakagawa, E. Y., Rashid, A., and Maldonado, J. C. (2010). Automating the mutation testing of aspect-oriented Java programs. In *AST'10*, pages 51–58. ACM.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Pattern, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[9] Horgan, J. R. and Mathur, A. P. (1992). Assessing testing tools in research and education. *IEEE Software*, 9(3):61–69.

[10] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[11] Kiczales et al., G. (1997). Aspect-oriented programming. In *ECOOP'97*, pages 220–242 (LNCS v.1241). Springer.

[12] Lacerda, J. T. S. and Ferrari, F. C. (2014). Towards the establishment of a sufficient set of mutation operators for AspectJ programs. In *SAST'14*, volume 2, pages 21–30. Brazilian Computer Society.

[13] Nakagawa, E. Y., Simão, A. S., Ferrari, F. C., and Maldonado, J. C. (2007). Towards a reference architecture for software testing tools. In *SEKE'07*, pages 157–162.

[14] Offutt et al., J. (1996). An experimental determination of sufficient mutant operators. *ACM TOSEM*, 5(2):99–118.

[15] Vincenzi, A. M. R. (2004). *Object-oriented: Definition, Implementation and Analysis of Validation and Testing Resources*. PhD thesis, ICMC/USP, São Carlos, SP - Brazil.

[16] Vincenzi, A. M. R., Wong, W. E., Delamaro, M. E., and Maldonado, J. C. (2003). Jabuti: A coverage analysis tool for java programs. In *SBES'03*, pages 79–84.