Automating the Mutation Testing of Aspect-Oriented Java Programs

Fabiano Cutigi Ferrari
Computer Systems Department
University of São Paulo - Brazil
ferrari@icmc.usp.br

Elisa Yumi Nakagawa
Computer Systems Department
University of São Paulo - Brazil
elisa@icmc.usp.br

Awais Rashid
Computing Department
Lancaster University - UK
marash@comp.lancs.ac.uk

José Carlos Maldonado
Computer Systems Department
University of São Paulo - Brazil
jcmaldon@icmc.usp.br

ABSTRACT

Aspect-Oriented Programming has introduced new types of software faults that may be systematically tackled with mutation testing. However, such testing approach requires adequate tooling support in order to be properly performed. This paper addresses this issue, introducing a novel tool named *Proteum/AJ*. *Proteum/AJ* realises a set of requirements for mutation-based testing tools and overcomes some limitations identified in previous tools for aspect-oriented programs. Through an example, we show how *Proteum/AJ* was designed to support the main steps of mutation testing. This preliminary use of the tool in a full test cycle provided evidences of the feasibility of using it in real software development processes and helped us to reason about the current functionalities and to identify future needs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Verification, Measurement

Keywords

Mutation testing, Aspect-Oriented Programming, test automation, testing tools.

1. INTRODUCTION

In the last years, Aspect-Oriented Programming (AOP) [18] has been widely investigated as an approach for enhancing software modularity. It has introduced new concepts and programming mechanisms that allow software developers to implement different system functionalities separately. Once implemented, they can be combined together to produce a single executable system that is expected to present enhanced modularity and maintainability [19].

Despite the benefits claimed by the AOP community, the associated programming mechanisms lead to specific types of faults [13, 14]. In this context, mutation testing [12] is a fault-based testing criterion that can systematically explore common mistakes made by aspect-oriented (AO) software developers. It has been widely investigated in several development phases and technologies [17] and recently in AOP field [8, 10, 13].

Mutation testing strongly relies on automated testing tools in order to be introduced in real software development environments. Moreover, testing tools are invaluable resources for research and education, as highlighted by Horgan and Mathur [16]. Examples of mutation-based testing tools that have been used with success by software practitioners in both industry and academia are the *Mothra* [9] and *Proteum* [11] tools. In regard to AO software, we can also identify some attempts to provide tooling support for mutation testing [8, 10]. They all focus on programs written in AspectJ [7], which is the most widely adopted AOP language. Despite that, none of these tools currently provide adequate support to the basic steps performed in mutation testing.

In our previous research we investigated how fault types may occur in AO software and how this can be addressed with mutation testing [13]. In this paper we present *Proteum/AJ*, a novel tool for mutation testing of AspectJ AO programs. *Proteum/AJ* automates a set of AO-specific mutation operators [13] and supports the basic steps of mutation testing [12]. It leverages previous knowledge on developing *Proteum* (**Pro**gram **Te**sting **U**sing **M**utants) [21], a family of tools for mutation testing developed by the Software Engineering group at the University of São Paulo, Brazil.

Proteum/AJ aims at filling the gap of limited support for mutation testing of AO programs by providing a set of functionalities not yet available in current tools. The development of *Proteum/AJ* was guided by a set of basic requirements for mutation tools mostly identified from previous research (Section 2). For instance, it allows the tester to perform incremental testing by evolving the selection of mutation operators, target aspects and the test suite. Equivalent mutants can be automatically identified, and live mutants can be individually executed and analysed.

The tool implements a reference architecture for software testing tools named *RefTEST* [22], from which the main functional modules were derived (Section 3). The preliminary use of the tool (Section 4) evinces the feasibility of using *Proteum/AJ* in real-world software development processes. Moreover, using the tool provided us with valuable feedback with respect to the implemented functionalities and issues to be addressed in our future research (Sections 5 and 6).

AST '10, May 3-4, 2010, Cape Town, South Africa

Table 1: Requirements for mutation-based testing tools.

Requirement	Description
1. Test case handling*	Execution, inclusion/exclusion, activation/deactivation of test cases.
2. Mutant handling*	Creation, selection, execution and analysis of mutants.
3. Adequacy analysis*	Computation of mutation score generation of statistical reports.
4. Reducing test costs†	Auto-generation of test cases and minimisation of test sets.
5. Unrestricted program size†	Size of the target application should not be restricted by the tool.
6. Support for testing strategies⊙	Different testing strategies (e.g. ordered application of mutation operators) should be supported by the tool.
7. Independent test configuration	The test configuration (e.g. test inputs, outputs and executing environment) should not be restricted by the tool.

*From Delamaro and Maldonado [11]. ⊙From Vincenzi et al. [23]. †From Horgan and Mathur [16].

2. BACKGROUND AND RELATED WORK

Mutation testing is a fault-based testing criterion which relies on common mistakes practitioners make during software development [12]. Such mistakes are modelled into *mutation operators* as a set of transformation rules. Given an original product[1] P, the criterion requires the creation of a set M of *mutants* of P, resulting from the application of the mutation operators to it. Then, for each mutant $m, (m \in M)$, the tester runs a test suite T originally designed for P. If $\exists t, (t \in T) \mid m(t) \neq P(t)$, this mutant is considered *killed*. If not, the tester should improve T with a test case that reveals the difference between m and P. If m and P are equivalent, then $P(t) = m(t)$ for all test cases.

A recent survey undertaken by Jia and Harman [17] showed that mutation testing has been extensively and increasingly investigated in the last three decades. According to their survey, over 30 tools have been implemented to automate mutation testing at several levels of software abstraction, ranging for formal specification to source code level testing. Specifically regarding source code testing, *Mothra* [9] for Fortran programs and *Proteum* [11] for C programs are the two most widely investigated and used tools.

From the experience in developing and using *Proteum*, Delamaro and Maldonado [11] listed a minimal set of requirements that should be provided by mutation-based testing tools. Basically, their list regards test case handling, mutant handling and adequacy analysis. We enhanced this list of requirements with some common features observed by Horgan and Mathur [16] in a suite of testing tools used in their experiments, including *Mothra*. The resulting list is shown in Table 1.

Recently, Vincenzi et al. [23] proposed the *Muta-Pro* process to support mutation-based testing. It aims at synergically integrating several approaches to reduce the high cost of mutant execution and analysis tasks. The process was designed to support experimentation in software testing, for instance, supporting incremental testing strategies along the process. From *Muta-Pro* we identified an extra requirement that is listed in Table 1. Moreover, we included the *Independent test configuration* feature, which we consider fundamental for testing modern software systems. For such systems (e.g. enterprise information systems, frameworks and software product lines), the tool should not restrict, for instance, the test input and test output formats, neither the configuration of the executing environment. Such properties should be delegated for specific test execution mechanisms whose outputs (e.g. results and reports) could be extracted and analysed by the testing tool.

[1] P can be a program specification, source code or any other executable software artefact [23].

2.1 Current Mutation Tools for AO Programs

We identified two implementations of mutation tools for AO software, both for AspectJ programs. Anbalagan and Xie [8] implemented a tool that performs mutations of pointcut expressions. Mutants are produced through the use of wildcards as well as by using naming parts of original pointcuts and join points identified from the base code. Based on heuristics, the tool automatically ranks the most representative mutants pointcuts, which are the ones that more closely resemble the original pointcuts. If a mutant selects the same set of join point as does the original expression, it is automatically classified as equivalent. The final output is a list of the ranked mutants.

Anbalagan and Xie [8]'s tool has some limitations[2] with respect to the requirements presented in Table 1. It does not support all steps of mutation testing. Basically, the tool is limited to the creation and classification of mutants based on a very small set of mutation operators. No support for test case and mutant handling is provided. Other particular limitation regards the equivalent mutant detection. In AspectJ, pointcuts can be reused in different modules, hence several join point matchings across the system may be affected by a single pointcut mutation. However, the tool analyses pointcuts individually, potentially overlooking the impact of a mutation on other modules.

More recently, Delamare et al. [10] presented the *AjMutator* tool that implements a subset of the operators for pointcut expressions proposed in our previous research [13]. *AjMutator* parses pointcut expressions from aspects individually and performs the mutations. The modified expressions are reinserted into the code, generating the mutants. The automatic detection of equivalent mutants relies on join point matching information provided by the abc compiler [1], an alternative compiler for AspectJ programs. *AjMutator* allows the tester to run JUnit test cases and identifies non-compilable and dead mutants. The tool outputs an XML file that contains information about every mutant handled (e.g. mutant status, pointcut ID and aspect ID).

Although *AjMutator* tackles some of the limitations observed in Anbalagan and Xie's tool [8], it still misses some basic functionalities to properly support mutation testing. For instance, it does not allow for mutation operator selection, hence hindering testers to apply different strategies. The mutant analysis itself is limited to the automatic detection of equivalent mutants; other mutant handling features such as individual mutant execution and manual classification of mutants are not available. Specific implementation issues include the lack of support to Java 5 features [6] in the

[2] Since the tool is not available for download, our analysis is based on the description provided by the authors [8].

Table 2: Limitations of current tools for mutation testing of AO programs.

Req.	AjMutator [10]	Anbalagan and Xie's tool [8]
#1	Supports test case execution. However, tests cannot be incrementally added, activated or deactivated.	No support for test case management and execution.
#2	All implemented operators are applied at once. Does not support manual mutant inspection and individual mutant execution.	All implemented operators are applied at once. No support for mutant handling is provided.
#3	Equivalent mutants are automatically identified. Calculates mutation score but does not create statistical reports.	Equivalent mutants are automatically identified. Does not compute mutation score neither creates statistical reports.
#4	No support for test case generation or minimisation of test suites.	No support for test case generation or minimisation of test suites.
#5	n/a	n/a
#6	No support for testing strategies (e.g. incremental use of operators or addition of test cases).	No support for testing strategies (e.g. incremental use of operators or addition of test cases).
#7	Test execution is configured through Ant tasks (i.e. it is delegated to external tools).	No support for test execution.

abc compiler, as well as the lack of support for incremental testing since the results are reset in every test run.

A summary of the addressed requirements and limitations of these tools is presented in Table 2. The requirement numbers are presented in the first column and "n/a" means that we could not find such information in the original work. In the next sections we describe how we designed the *Proteum/AJ* tool to overcome some of the current limitations.

3. THE *PROTEUM/AJ* TOOL

Proteum/AJ is a tool for mutation testing of AO Java programs written in AspectJ. Table 3 summarises up front how *Proteum/AJ* addresses the requirements for mutation tools listed in Section 2. It also compares our tool with the two previous tools described in Section 2.1. In short, *Proteum/AJ* supports the four main steps of mutation testing, as originally described by DeMillo et al. [12]: (i) the original program is executed on the current test set and test results are stored; (ii) the mutants are created based on a mutation operator selection that may evolve in new test cycle iterations; (iii) the mutants can be executed all at once or individually, as well as the test set can be augmented or reduced based on specific strategies; and (iv) the test results are evaluated so that mutants may be set as dead or equivalent, or mutants may remain alive.

Table 3: Addressed requirements.

Requirement	*Proteum/AJ*	*AjMutator*	Anbalagan
1. Test case handling	partial	partial	no
2. Mutant handling	yes	partial	partial
3. Adequacy analysis	partial	partial	partial
4. Reducing test costs	no	no	no
5. Unrestricted program size	yes	n/a	n/a
6. Support for testing strategies	partial	no	no
7. Independent test configuration	yes	yes	no

More details about how requirements are fully or partially addressed by *Proteum/AJ* are further discussed in Section 5. Next we present an overview of the tool architecture (Section 3.1) and its data model (Section 3.2). The main supporting technologies and implementation details are presented in Sections 3.3 and 3.4, respectively.

3.1 Architecture

Figure 1 depicts the architecture of *Proteum/AJ*. The architecture is based on a reference architecture for software testing tools called *RefTEST* [22]. *RefTEST* is based on separation of concerns (SoC) principles, the Model-View-Controller (MVC) and three tier architectural patterns, and the ISO/IEC 12207 standard for Information Technology. *RefTEST* encourages the use of aspects as the mechanism for integrating the core activities of a testing tool with tools that automates supporting and organisational software engineering activities defined in ISO/IEC 12207 (e.g. planning, configuration management and documentation tools). Moreover, aspects are also encouraged for integrating services such as persistence and access control. *Proteum/AJ* benefited mainly from the reuse of domain knowledge contained in *RefTEST*. The instantiation of the architecture provided us with guidance on how we could structure the tool in terms of functionalities and module interactions.

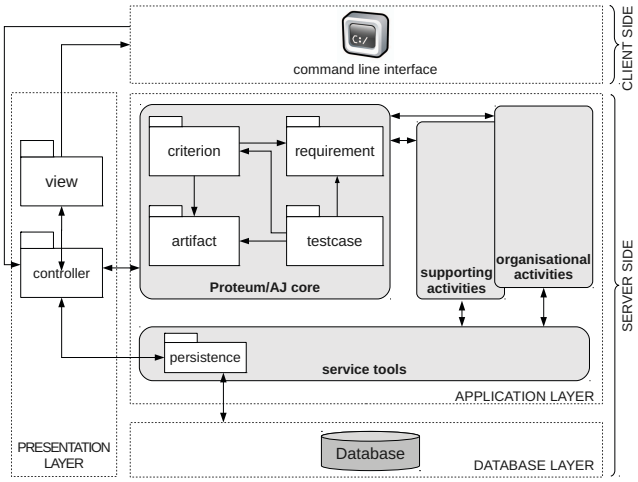

Figure 1: *Proteum/AJ* architecture.

The modules currently implemented in *Proteum/AJ* are shown as UML packages in Figure 1. The core of the tool comprises the four main concepts that should be handled by testing tools, as proposed in *RefTEST* [22]: *testing criterion*, *artifact*, *test requirement* and *test case*. Some of them map directly to the requirements presented in Table 1. For instance, testcase maps to the *"test case handling"* requirement; and criterion maps to both *"mutant handling"* and *"adequacy analysis"* requirements. The former provides functionalities for running and managing test cases in *Proteum/AJ*, while the latter is responsible for handling tasks related to testing criterion itself (e.g. generating, compiling and analysing mutants).

The artifact and requirement modules comprise, respectively, the artefacts under test (i.e. the AspectJ source code files) and the test requirements (i.e. the generated mutants). The controller module is in charge of receiving requests from the client and properly invoking the modules present in the application layer, which include core functionalities and database-related procedures. The controller

is also responsible for updating the view that is presented to the client. In *Proteum/AJ*, the view is basically formed by test execution feedback that is displayed to the users.

3.2 Data Model

Figure 2 shows the data model of *Proteum/AJ* using the Extended Entity-Relationship notation. In the *Proteum/AJ* database, a TestProject element represents a test project and belongs to a user (a User element in Figure 2). Each TestProject selects one or more AspectJ source code files (represented by the SourceCode) that are the targets of one or more mutation operators (MutationOperatorBean element). The mutations implemented by the mutant operators are performed on the SourceCode elements, resulting in a set of mutants (Mutant element). Each TestProject also executes a set of test cases (AntTestCase element), which are also executed against the mutants within the same test project.

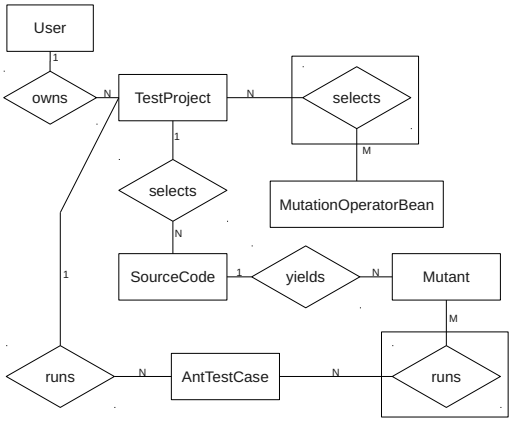

Figure 2: *Proteum/AJ* data model.

3.3 Supporting Technologies

We employed a set of technologies to implement *Proteum/AJ* functionalities. The main ones are:

AspectJ-front: In *Proteum/AJ*, parsing and pretty-printing of AspectJ source code are supported by the AspectJ-front tools suite [3]. The AspectJ-front parser converts AspectJ code into an abstract syntax tree (AST) represented in aterm notation. aterm is a format for exchanging structured data between tools [4]. Figure 3 exemplifies how an `after returning` AspectJ advice (top part) is represented in aterm notation (bottom part).

Original advice in AspectJ

```
after(Customer caller, Customer receiver, boolean iM)
   returning(Connection c) : createConnection(caller, receiver, iM) {
   ...
}
```

The same advice in aterm notation

```
AdviceDec([],
 After([Param([],ClassOrInterfaceType(TypeName(
          Id("Customer")),None()),Id("caller")),
        Param([],ClassOrInterfaceType(TypeName(
          Id("Customer")),None()),Id("receiver")),
        Param([],Boolean(),Id("iM"))],
        Some(Returning(Param([],ClassOrInterfaceType(
          TypeName(Id("Connection")),None()),Id("c")))),
 None(),
 NamedPointcut(PointcutName(Id("createConnection")),[ ... ]),
 Block([ ... ])
)
```

Figure 3: Example of aterm representation.

iBATIS: We used the iBATIS data mapper framework [5] in *Proteum/AJ* to handle data storage. With iBATIS, persistent objects are mapped to database table by means of XML-based procedures that can be invoked with pure Java code. The framework also provides full support for database connection pooling, caching and transactions.

Ant: The Ant tool [2] provides facilities for compiling and manipulating files in build processes through a set of XML-based procedures called *tasks*. Such tasks can be programmatically invoked, what facilitates the integration of Ant with customised tools such as *Proteum/AJ*. Examples of Ant tasks used within *Proteum/AJ* are AspectJ source code compilation (iajc task), test case execution (junit task) and file decompression (unzip task).

3.4 Implementation Details

This section describes the main modules of the *Proteum/AJ* core: the criterion and the testcase modules. A simplified class diagram of these modules is depicted in Figure 4. Details are provided in the sequence.

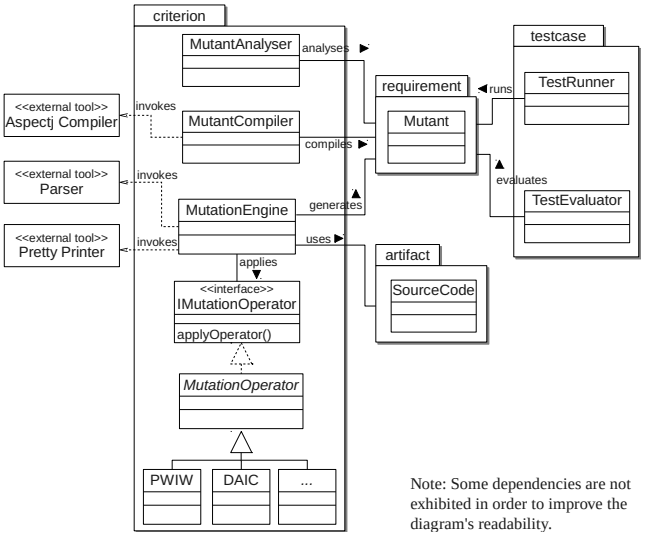

Figure 4: *Proteum/AJ* core.

3.4.1 *The* criterion *Module*

The criterion module includes functionalities for generating, compiling and analysing mutants. These functionalities are implemented within three main classes: `MutationEngine`, `MutantCompiler` and `MutantAnalyser`.

The `MutationEngine` class is responsible for generating mutants from AspectJ source code by controlling the application of the mutation operators. The operators implemented in *Proteum/AJ* were proposed in our previous research [13]. They model faults that are likely to occur in AspectJ-like programs. These operators are summarised in Table 4. They are split into three groups, according to the main AOP constructs[3] they are related to, namely: Group G1 – pointcut-related operators; Group G2 – `declare`-like-related operators; and Group G3 – advice-related operators. All mutation operators implement the `IMutationOperator` interface, what facilitates the inclusion of new mutation operators into the tool. Mutations are performed in the aterm representation of source code elements, which are manipulated as Java `String`

[3]More information about the main AOP constructs can be found in the AspectJ documentation [7]

Table 4: Mutation operators implemented in Proteum/AJ (adapted from [13]).

Operator		Description/Consequences
G1	PWSR†	Pointcut weakening by replacing a type with its immediate supertype in pointcut expressions
	PWIW*	Pointcut weakening by inserting wildcards into pointcut expressions
	PWAR*	Pointcut weakening by removing annotation tags from type, field, method and constructor patterns
	PSSR†	Pointcut strengthening by replacing a type with its immediate subtype in pointcut expressions
	PSWR*	Pointcut strengthening by removing wildcards from pointcut expressions
	PSDR*	Pointcut strengthening by removing `declare @` statements, used to insert annotations into base code elements
	POPL*?	Pointcut weakening or strengthening by changing parameter lists of primitive pointcut designators
	POAC*?	Pointcut weakening or strengthening by changing `after [retuning|throwing]` advice clauses
	POEC*	Pointcut weakening or strengthening by changing exception throwing clauses
	PCTT*?	Pointcut changing by replacing a `this` pointcut designator with a `target` one and vice versa
	PCCE	Context changing by switching `call/execution/initialization/preinitialization` pointcuts designators
	PCGS*	Pointcut changing by replacing a `get` pointcut designator with a `set` one and vice versa
	PCCR*?	Pointcut changing by replacing individual parts of a pointcut composition
	PCLO*	Pointcut changing by changing logical operators present in type and pointcut compositions
	PCCC*?	Pointcut changing by replacing a `cflow` pointcut designator with a `cflowbelow` one and vice versa
G2	DAPC	Aspect precedence changing by alternating the order of aspects involved in `declare precedence` statements
	DAPO	Arbitrary aspect precedence by removing `declare precedence` statements
	DSSR	Unintended exception handling by removing `declare soft` statements
	DEWC	Unintended control flow execution by changing `declare error/warning` statements
	DAIC	Unintended aspect instantiation by changing `perthis/pertarget/percflow/percflowbelow` deployment clauses
G3	ABAR	Advice kind changing by replacing a `before` clause with an `after [retuning|throwing]` one and vice versa
	APSR	Advice logic changing by removing invocations to `proceed` statement
	APER	Advice logic changing by removing guard conditions which surround `proceed` statements
	AJSC	Static information source changing by replacing a `thisJoinPointStaticPart` reference with a `thisEnclosingJoinPointStaticPart` one and vice versa
	ABHA	Behaviour hindering by removing implemented advices
	ABPR	Changing pointcut-advice binding by replacing pointcuts which are bound to advices

†Not implemented in the current version of *Proteum/AJ*.
*Considered for automatic detection of equivalent mutants. ?Impacts quantification of join point with dynamic residues.

objects by the `MutationEngine` class. The engine uses a set of tailor-made string handlers that enables such manipulation (e.g. code offset localising and replacement).

The `MutationEngine` class also invokes the AspectJ-front pretty-printer tool to build AspectJ code. This tool performs a first check that ensures each mutant code is syntactically correct when considered in isolation, i.e. before the aspect is woven into the base code. However, it cannot guarantee that base code and the mutant aspect can be successfully woven together. Such verification can only be performed by an AspectJ compiler such as ajc [7].

Invoking the ajc compiler is assigned to the `MutantCompiler` class. It is achieved through the iajc Ant task provided with the AspectJ API. Non-compilable mutants are identified at this step. Such mutants are classified as *anomalous* and are further discarded when the mutation score is calculated.

Finally, the `MutantAnalyser` class compares the original and the mutated code. More specifically, it automatically identifies equivalent mutants for some pointcut-related operators based in the weaving output produced by the compiler. This output is a valuable information that includes details of all matching between aspects and base code [7]. The `MutantAnalyser` also generates reports that show the modified portions of code for each mutant, for each target aspect. These reports are necessary to help the testers identify either equivalent mutants or the need for additional test cases.

3.4.2 *The* `testcase` *Module*

The role of the testcase module is managing test execution and evaluation. Within it, the `TestRunner` class implements test runner methods for the original and the mutant applications. In particular, it calls the `MutantCompiler` class to produce the executable mutant application. If the compilation succeeds, the automatic detection of equivalent mutants is performed by the `MutantAnalyser` class[4]. If the original application and the mutant are not equivalent, the tests are executed and the results are stored for further evaluation.

The `TestEvaluator` class evaluates the test results obtained from the execution of the original application and the mutants. It contrasts test case outputs, identifies the differences and decides whether a given mutant should be killed or not.

4. EXAMPLE

This section describes the use of *Proteum/AJ* based on the modules presented in the previous section. It starts presenting an example (Section 4.1) that is used along the section to exemplify the main steps performed with *Proteum/AJ*. In the sequence, we show the results obtained with this example along a full test session.

4.1 The Telecom Application

To introduce the *Proteum/AJ* functionalities, we selected an AspectJ application called Telecom. It is a telephony system simulator which is originally distributed with AspectJ [7]. In Telecom, timing and billing of phone calls are handled by aspects. The version we use in this example includes six classes and three aspects. It extends the original version to support a different type of charging for mobile calls [20].

Figure 5 shows the partial implementation of the three aspects present in Telecom. The `Timing` aspect measures the duration of the calls, which are logged by the `TimerLog` aspect. `Billing` implements the billing concern and ensures calls are charged accordingly. Note that Figure 5 only shows AspectJ code that is relevant for the mutation operators implemented in *Proteum/AJ*, i.e. pointcuts, advices and `declare`-like statements. The full implementation can be found at `http://www.labes.icmc.usp.br/~ferrari/proteumaj/`.

[4]More details about equivalent mutant detection in Section 4.

```
public aspect Timing {
  ...
  after(Connection c) returning : target(c) &&
        call(void Connection.complete()) {
    getTimer(c).start();
  }

  pointcut endTiming (Connection c) :target(c) &&
        call(void Connection.drop());

  after(Connection c) returning : endTiming(c) {
    getTimer(c).stop();
    c.getCaller().totalConnectTime += getTimer(c).getTime();
    c.getReceiver().totalConnectTime += getTimer(c).getTime();
  }
}


public aspect TimerLog {
  after(Timer t) returning : target(t) && call(* Timer.start()) {
    System.err.println("Timer started: " + t.startTime);
  }

  after(Timer t) returning : target(t) && call(* Timer.stop()) {
    System.err.println("Timer stopped: " + t.stopTime);
  }
}


public aspect Billing {
  declare precedence : Billing, Timing ;
  ...
  pointcut createConnection (Customer caller, Customer receiver,
           boolean iM):
       args(caller, receiver, iM) && call(Connection+.new(..) );

  after(Customer caller, Customer receiver, boolean iM) returning(Connection c):
        createConnection(caller, receiver, iM) {
    if(receiver.getPhoneNumber().indexOf("0800") == 0)
      c.payer = receiver;
    else
      c.payer = caller;
    c.payer.numPayingCalls += 1;
  }
  ...
  after(Connection conn) returning: Timing.endTiming(conn) {
    long time = Timing.aspectOf().getTimer(conn).getTime();
    long rate = conn.callRate();
    long cost = rate * time;
    if(conn.isMobile()) {
      if(conn instanceof LongDistance) {
        long receiverCost = MOBILE_LD_RECEIVER_RATE * time;
        conn.getReceiver().addCharge(receiverCost);
      }
    }
    getPayer(conn).addCharge(cost);
  }
  ...
  void around(boolean isMobile): execution(Connection.new(.., boolean) ) &&
        args(.., isMobile) {
    System.err.println("The established Connection includes a mobile device");
    if(isMobile)
      proceed(isMobile);
    else {
      proceed(isMobile);
    }
  }
} // end of Billing
```

Figure 5: Partial code of the Telecom application.

4.2 Testing with Proteum/AJ

As introduced early in Section 1 as well in this section, *Proteum/AJ* supports the main steps performed in a typical mutation-based testing process. Figure 6 depicts how this is achieved with the tool. It shows the execution sequence of the main modules and the inputs/outputs of each of them. The modules are invoked through parameterised scripts that are executed via command line in a shell console.

Pre-processing the original application

As shown in Figure 6, the target application must be a compressed file that is submitted to *Proteum/AJ*. This file contains all modules (classes, aspects and libraries) of the application under test. The Application Handler module then runs a pre-processing step, whose outputs are the decompressed original application and a list of target aspects. This step creates the test projects in the *Proteum/AJ* database, according to the schema presented in Figure 2. Such data is handled along the test process. The Application Handler also compiles the original application through the iajc Ant task.

Execution of the original application

The decompressed application is sent to the Test Runner module together with the test case files. The Test Runner executes the application on the available test set by invoking the junit Ant task. The results are stored for further evaluation of mutants.

We designed an initial test set that includes 18 test cases that target all modules of the Telecom system. This test set covers all control flow- and data flow-based requirements computed according to the approach for structural testing of AO programs proposed by Lemos et al. [20]. It took ∼1.6 second to execute the 18 tests in *Proteum/AJ*.

Generation of mutants

The Mutation Engine receives as input the list of target aspects identified by the Application Handler and the set of mutation operators selected by the tester. It produces the set of mutants that are passed to the Mutant Compiler.

The 24 operators implemented in *Proteum/AJ* produced a total of 111 mutants for the three aspects of Telecom in ∼1.8 second. Table 5 summarises the results. The mutants are grouped according to the nature of the mutation operators (see Table 4).

Table 5: Telecom mutants.

	Group G1	Group G2	Group G3	Total
Billing	30	2	15	47
TimerLog	26	0	6	32
Timing	26	0	6	32
Total	82	2	27	111

Execution of mutants

The execution of mutants requires a series of steps in *Proteum/AJ*. Initially, each mutant is sent to the Mutant Compiler module. This module invokes the ajc compiler through the iajc Ant task provided with the AspectJ API [7]. The Mutant Compiler detects non-compilable mutants which are classified as *anomalous*. For compilable mutants, the weaving information produced by the ajc compiler is collected at this stage and further used by the Mutant Analyser module.

The Mutant Analyser compares the ajc weaving output of the original application and the mutants. *Proteum/AJ* uses such information to decide whether mutants created by a subset of operators from Group G1 (tagged with the symbol "⋆" in Table 4) are equivalent to the original aspects. The join point quantification yielded by these operators can be checked at compilation time so that equivalence can be automatically obtained. However, some pointcut designators (e.g. `if` and `cflow`) results in dynamic tests being inserted into the affected join points during the weaving process. These tests, also called *dynamic residues* [15], imply that final join point matching can only be resolved at runtime.

We identified mutation operators that are likely to impact the quantification of join points that contain dynamic residues (tagged with the symbol "?" in Table 4). *Pro-*

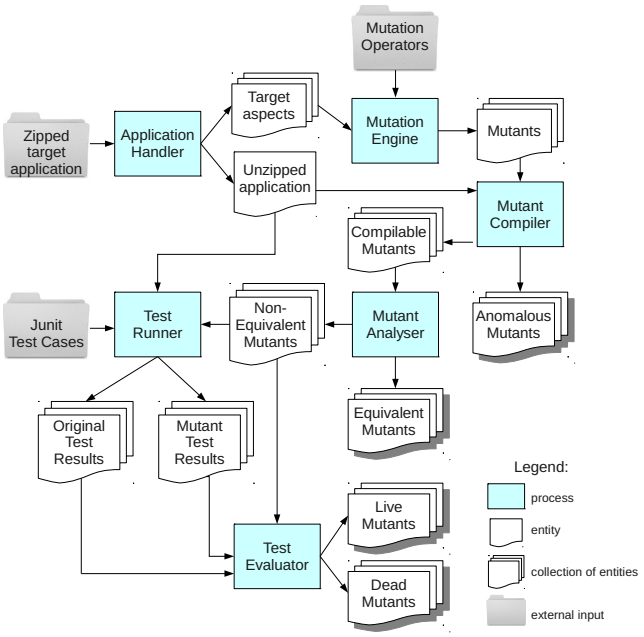

Figure 6: *Proteum/AJ* execution flow.

teum/AJ, in turn, gives the tester the option of deactivating the automatic equivalence detection by setting a flag sent to the Mutant Compiler module. In doing so, the tester becomes in charge of manually analysing the mutants created by those operators during the *Analysis of mutants* step.

As for the original application, the Test Runner module runs all compilable, non-equivalent mutants on the current test set using the iajc Ant task. The results are stored and sent to the Test Evaluator module. The evaluator contrasts the results with the original results and identifies which mutants should be killed. Mutants can also be killed by execution timeout. In this case, the timeout is parameterised either within the junit Ant task or via console script.

Table 6: Results for Telecom after running the tests.

	Alive	Dead	Equivalent*	Anomalous	Total
`Billing`	8	13	14	12	47
`TimerLog`	16	0	16	0	32
`Timing`	10	6	16	0	32
Total	34	19	46	12	111

*automatically identified.

According to the numbers presented in Table 6, 99 mutants of Telecom compiled with success and 12 were classified as anomalous. The execution of the original test set took ∼3 minutes (or ∼1.62 second per mutant), resulting in 19 dead mutants and 46 automatically identified as equivalent.

Analysis of mutants

This is the most costly step performed in mutation testing since it requires manual intervention from the tester. To support it, the Mutant Analyser module creates a set of plain text reports that show the differences between the original and mutated code. These reports show the modified parts of the code for each mutant, hence providing guidance for the tester while analysing the changes. The tester then decides which mutants should be set as equivalent and updates their status through the Mutant Analyser module.

The Mutant Analyser also computes the mutation score using the formula below, where dm is the number of dead mutants, cm is the number of compilable mutants and em is the number of equivalent mutants:

$$MS = (dm)/(cm - em)$$

The initial tests yielded in a mutation score of 0.3585. We analysed the 34 live mutants, what resulted in 12 mutants identified as equivalent. The remaining ones were killed by five extra test cases that we added to the initial test set. The final results are shown in Table 7. The final test set includes 23 test cases that lead to a mutation score of 1.0.

Table 7: Final results for Telecom.

	Alive	Dead	Equivalent	Anomalous	Total
`Billing`	0	19	16	12	47
`TimerLog`	0	12	20	0	32
`Timing`	0	12	20	0	32
Total	0	43	56	12	111

5. DISCUSSION

This section analyses how *Proteum/AJ* addresses the requirements presented in Section 2. We also discuss limitations and issues related to the implementation of the tool.

How does Proteum/AJ fulfil the requirements?

Proteum/AJ allows the tester to manage mutants in several ways. For example, mutants can be created, recreated and individually selected for execution. The execution can also be restricted to live mutants only, and these can be manually set as equivalent and vice versa, that is, equivalent mutants can be reset as alive.

The size of the application under test is not constrained by the tool. Decompression and compilation tasks are delegated to third-party tools configured through Ant tasks. We have been experimenting *Proteum/AJ* with larger applications (e.g. nearly 200 classes and 40 aspects [14]) and the tool has shown to be able to handle larger sets of mutants (∼3,600 in total), despite the compilation issues discussed in the sequence. The test setup is also fully configurable through junit Ant tasks that are invoked by *Proteum/AJ*. In this way, there are only minor dependencies between the test execution configuration and the tool.

Proteum/AJ also enables the tester to import and execute new test cases within an existing test project, although it does not support test case activation/deactivation. Despite that, the *Proteum/AJ* database was designed to support such features in the future. The mutation score can be computed at any time after the first tests have been executed; however, the creation of statistical reports is another functionality that is planned for the upcoming releases.

Proteum/AJ implementation issues and limitations

Since *Proteum/AJ* runs JUnit test cases to evaluate the mutants, we can consider the tool implements the *firm mutation* approach. JUnit test cases have the ability of configuring partial program runs (e.g. a single method execution) and performing assertions in the course of the execution. Firm mutation is defined by Woodward and Halewood [24] as "*the situation where a simple error is introduced into a program and which persists for one or more executions, but not for the entire program execution*". Thus, similarly to the *AjMutator* tool [10], the evaluation of mutants based on partial program executions implemented in JUnit tests characterises firm mutation in *Proteum/AJ*.

Regarding the mutant compilation step, the ajc compiler allows for two types of weaving [7]: compile-time and post-

compile weaving[5]. The former is the simplest approach and is performed when the source code is available. The latter, on the other hand, is carried out for existing class files. In *Proteum/AJ*, the compiler directives are provided by the tester through the iajc Ant task. That is, the Ant task defines how the application will be compiled. So far we have only experienced *Proteum/AJ* with applications configured for weaving based on source code (compile-time). However, the compilation time may become a bottleneck for larger systems. Therefore, reducing the compilation time (e.g. through post-compile weaving) is one of the enhancements we planned for the next releases of the tool. Nevertheless, full weaving would still be required in the occurrence of inter-type declarations in the system [7], what is a very recurring situation we have noticed in complex AO systems [14].

To run *Proteum/AJ*, the compressed file submitted to *Proteum/AJ* is expected to include an Ant build file named build.xml. This build file must include three tasks that will be used by *Proteum/AJ*: (i) compile, which specifies how to compile the application; (ii) full-test, which specifies how to run the full test set (possibly) included with the application; and (iii) single-test, which specifies how to run a single JUnit test file. The tester can also provide a file named targets.lst within the compressed file. It contains a list of target aspects that is parsed by the Application Handler module (see Section 4). Optionally, target aspects and test case files can be added to the test project after it has been created.

6. FINAL REMARKS

In this paper we presented *Proteum/AJ*, a tool that automates the mutation testing of aspect-oriented AspectJ programs. The tool was planned to address some limitations of previous tools with the same intent [8, 10]. Its development was guided by a reference architecture for software testing tools [22] and by a set of requirements mostly identified from previous research on test automation [11, 16, 23]. The reference architecture established the main modules that compose the tool. On the other hand, the requirements defined the set of functionalities implemented within those modules.

We described *Proteum/AJ*'s characteristics through an example of use. The tool implements a set of mutation operators [13] that subsumes previous implementations [8, 10]. *Proteum/AJ* is able to generate and manage mutants for multiple aspects within a single test project. The mutant sets can be augmented or reduced along the test cycles, as well as the test suites that can be evolved in order to achieve higher test coverages (i.e. enhanced mutation scores). The example of use showed that employing *Proteum/AJ* in real software development processes is an achievable goal.

Our next research steps include the conduction of extra case studies that will comprise larger aspect-oriented systems. We also intend to fulfil the requirements not yet addressed in *Proteum/AJ*, specially the ones that regards test case handling and adequacy analysis. Besides, considering the complementary nature of testing approaches, we also aim to investigate how the tool can be integrated with other testing tools in order to share common resources such as test projects and test suites.

Acknowledgments

We would like to thank Martin Bravenboer from the Stratego/XT Team for the ready replies to doubts and required fixes in the AspectJ-front toolset. We also thank the financial support received from FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1), EC Grant AOSD-Europe (IST-2-004349), and CPNq - Brazil.

[5]A third type, the load-time weaving, is basically the post-compile weaving postponed to the moment classes are loaded to the JVM [7].

References

[1] abc: The aspectbench compiler for AspectJ. `http://abc.comlab.ox.ac.uk/` - accessed on 01/03/2010.

[2] Ant. `http://ant.apache.org/` - accessed on 01/03/2010.

[3] AspectJ-front. `http://strategoxt.org/Stratego/AspectJFront` - accessed on 01/03/2010.

[4] ATerm format. `http://www.program-transformation.org/Tools/ATermFormat` - accessed on 01/03/2010.

[5] iBATIS data mapper. `http://ibatis.apache.org/` - accessed on 01/03/2010.

[6] J2SE 5.0: New features and enhancements, 2004. `http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html` - accessed on 01/03/2010.

[7] AspectJ documentation, 2010. `http://www.eclipse.org/aspectj/docs.php` - accessed on 01/03/2010.

[8] P. Anbalagan and T. Xie. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In *ISSRE'08*, pages 239–248. IEEE Computer Society, 2008.

[9] B. J. Choi et al. The Mothra tool set. In *HICSS'89*, volume 2, pages 275–284, 1989.

[10] R. Delamare, B. Baudry, and Y. Le Traon. AjMutator: A tool for the mutation analysis of aspectj pointcut descriptors. In *Mutation'09 Workshop*, pages 200–204. IEEE Computer Society, 2009.

[11] M. E. Delamaro and J. C. Maldonado. Proteum: A tool for the assessment of test adequacy for C programs. In *PCS'96*, pages 79–95, 1996.

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

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

[14] F. C. Ferrari et al. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *ICSE'10*. ACM Press, 2010. (to appear).

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD'04*, pages 26–35. ACM Press, 2004.

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

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Tech.Report TR-09-06, CREST Centre, King's College, London - UK, 2009.

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

[19] R. Laddad. Aspect-oriented programming will improve quality. *IEEE Software*, 20(6):90–91, 2003.

[20] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[21] J. C. Maldonado et al. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation 2000 Symposium (Tool Session)*, pages 113–116. Kluwer, 2000.

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

[23] A. M. R. Vincenzi, A. S. Simão, M. E. Delamaro, and J. C. Maldonado. Muta-Pro: Towards the definition of a mutation testing process. *Journal of the Brazilian Computer Society*, 12(2):49–61, 2006.

[24] M. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Workshop on Soft. Testing, Verification, and Analysis*, pages 152–158. IEEE Computer Society, 1988.