mu2: A Refactoring-Based Mutation Testing Framework for Erlang

Ramsay Taylor, John Derrick

▶ **To cite this version:**

Ramsay Taylor, John Derrick. mu2: A Refactoring-Based Mutation Testing Framework for Erlang. 27th IFIP International Conference on Testing Software and Systems (ICTSS), Nov 2015, Sharjah and Dubai, United Arab Emirates. pp.178-193, 10.1007/978-3-319-25945-1_11 . hal-01470148

HAL Id: hal-01470148
https://inria.hal.science/hal-01470148

Submitted on 17 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

mu2: A refactoring-based mutation testing framework for Erlang

Ramsay Taylor and John Derrick

Department of Computer Science, The University of Sheffield

Abstract. We present a mutation testing framework for the Erlang functional programming language. Mutation testing evaluates a test set by mutating the original System Under Test (SUT) and measuring the test set's ability to detect the change. Designing mutation operators can be difficult, since they must modify the original program in a way that is both semantically significant, and a realistic simulation of a potential fault (either a fault with the system in its real context, or a common programmer error). The principal contribution of this work is the *mu2* framework, which leverages the Wrangler refactoring API to allow users to specify their own mutation operators. The framework makes it possible to quickly and clearly define mutation operators that can have complex and subtle effects on the SUT. This allows users to define domain-specific operators that can simulate faults that are of particular relevance to their project, rather than relying on standard operators. The mutation testing framework was evaluated in an industrial setting and compared to code coverage test adequacy metrics. It was found to be a valuable compliment to code coverage techniques, since it was able to uncover additional testing limitations that could not be easily identified by coverage alone.

1 Introduction

Testing is a vital component of any software development process, and often accounts for a large portion of the development effort. The purpose of testing is to provide assurance that the software functions correctly. However, as software projects expand, the size and complexity of the test sets also expand. This can create a new requirement to provide assurance of the "correct functioning" of the test set, i.e. that the test set is adequately assessing the software functionality.

Although measures such as code coverage provide some information about a test set's scope, they may not provide an accurate measure of a test set's ability to detect faults [8]. Mutation testing [10] provides an alternative approach, which has been shown to be able to identify limitations of test sets that could not be identified even with advanced coverage metrics such as MC/DC [4]. Mutation testing is a testing methodology which inserts deliberate faults into the System Under Test (SUT) to generate *mutants* of the program. The test set to be evaluated is run on each of these mutants. Since the tests were designed to evaluate the operation of the correct program they should report failure when run on the mutant. The mutation testing terminology is that the tests "kill" the

mutants. Those mutants that are not killed are either *semantically equivalent* to the original program — that is, although they will have undergone a syntactic change, they will have identical behaviour to the original — or they identify a class of fault that the test set is not adequately identifying. The percentage of mutants killed provides a numeric metric for the effectiveness of the test set.

Mutation testing has been applied successfully to various imperative languages, such as Java, C, and Ada [4, 11, 14, 7]. Simple, random changes to the syntax of the source files can produce many thousands of mutants easily, but a very high proportion will simply not compile, and many more will be semantically equivalent. An improvement over simple mutation testing is provided by first parsing the source file and then applying mutation operators to the parsed form — changing the semantics of the program directly — before re-rendering the program to a source code file.

It is not practical to seed every possible fault into a program and measure a test set's ability to detect these, so it is important that the mutants generated are in some way representative of a broader class of system faults. In this case a test set's ability to identify a particular deliberate fault provides good evidence that the test set is well written with respect to this class of fault or this particular section of the system. This provides some assurance that it would also identify other faults if they were present. Given this, it is important that the faults that are seeded in the mutants are representative of faults that are either likely or significant to the system under test. Consequently, while some general mutation operators are useful and provide a baseline measure of the quality of a test set, a principle objective of this work was to allow the development of domain-specific mutation operators for particular use cases. Specifically, we provide:

- A framework that allows the rapid development of semantically-rich mutation operators for specific domains
- Integration of mutation testing with the Erlang ecosystem and automation of a mutation testing workflow for Erlang modules
- An evaluation of the mutation testing framework with an industrial partner that demonstrates mutations testing's value, but also how it can be used as a *compliment* to other test adequacy metrics

The paper is structured as follows: Section 2 contains some background on Erlang and mutation testing. It also describes the process of applying mutation testing to the kinds of test suites common in Erlang. Section 3 describes the *mu2* framework that implements and automates that application of mutation testing to Erlang. Section 4 details our refactoring-based system for defining mutation operators that allows semantically rich operators to be developed rapidly. Section 5 documents the evaluation study carried out with Interoud Innovation. Finally, Section 6 concludes.

2 Background

2.1 Erlang

Erlang [2, 1, 15] is a programming language originally developed at Ericsson for use in their telecoms infrastructure products. It is now available as open source software, and used in a wide variety of companies both large and small. As a language it is declarative and uses several components of the functional programming paradigm, such as pattern matching and extensive use of recursion.

```
-module(abiftest).
-export([dv/2]).

dv(A,B) ->
    if (A == 0) and (B > 4) ->
            B;
       true ->
            B / A
    end.
```

Fig. 1. The *abiftest* erlang module

An Erlang module contains a number of functions, each of which is defined by a series of patterns starting with a name, a set of parameters, the arrow symbol `->`, the function definition, and ends with a full stop. For example, the `abiftest` module in Figure 1 contains just one function, `dv`, which takes two numbers A and B and divides B by A unless some conditions hold. Functions can have multiple patterns, with separate patterns separated by semicolons and the final pattern terminated with a full stop. The same syntax extends to internal decisions, such as the `if` statement in Figure 1, or `case` statements that match structural patterns over values. Patterns are matched in order with the first matching pattern being applied — hence the `if` statement having `true` as the final pattern, since this will always match and so functions like an `else` or `otherwise` in other languages.

Variable names begin with capital letters or the underscore character (e.g. `Var, _S`), whilst lower case letters indicate an "atom" value (conceptually a user defined keyword, e.g. `lock, unlock`). Tuples are contained in curly brackets (`{lock, S}`), lists in square brackets (`[a,b,c]`). Strings are treated as lists but can be presented in double quotation marks. Erlang is an interpreted language and so the failure of the interpreter to find a matching pattern for a particular function application is reported at runtime. There is an exception throwing model for error handling, which allows pattern matching over the types of exceptions caught.

Erlang also features a process-oriented distributed programming model that uses asynchronous communication channels. Messages sent from one process to another accumulate in the receiver's message queue. The `receive` construct allows the process to pattern match over the incoming messages. The first message is compared to the patterns and, if it matches any one, then the relevant code is executed. In the event that the first message in the queue does not match any of the patterns in the current `receive` construct, then the second message in the queue is compared, and so on until a message matches. In this way it is possible for an Erlang process to skip some messages and handle particular message patterns with higher or lower priority. In the event that no message matches the current patterns the process with block until either a message arrives that does match, or a time limit (specified with the `timeout` pattern) is reached.

There is a large range of testing frameworks and support for Erlang. Conventional unit testing is often carried out with the Eunit [5] framework, but there is considerable use made of more advanced test generation and property-based testing using the Erlang QuickCheck system [3].

2.2 Mutation Testing

The objectives of any test adequacy metric are:

- Give a general quality metric for the test set
- Identify specific weaknesses of the test set
- Give constructive feedback that guides a user to improve the test set and address the weaknesses

Mutation testing — first described in [10] — seeks to evaluate test sets by simulating faults in a software system and measuring the test set's ability to identify the faults. Standard mutation testing makes a modification to the software's source code to produce a 'mutant'. The mutant code is compiled and then tested using the test suite. If the mutant fails the test suite, such a mutant is referred to as 'killed', if not then it is 'alive'. Where a mutant remains alive it must be inspected to determine whether the mutation actually produced a functional change. In some cases the mutations to the source code have no effect on the semantics - changing the name of an unused variable, for example.

For a non-trivial program, it is not realistic to explore all possible mutations. This is why one would usually focus on those that seem 'representative' of the defects a program may contain. For example, a typical hypothesis here is of a 'competent programmer' who may introduce an occasional error (such as in a form of a comparison operator the wrong way around). In this case, a good test suite is the one that kills all single-comparison mutants.

For example, consider the simple test set given in Figure 2, which tests the *abiftest* module from Figure 1. The *abiftest* module passes all these tests, however, it contains a defect that is not identified by this test set. Specifically: the `dv` function can produce a divide by zero error if it is called in such a way that the first `if` decision is false but A contains the value 0. Because the decision is a

```
-module(abiftest_tests).

-include_lib("eunit/include/eunit.hrl").

zero_test() ->
        ?assert(abiftest:dv(5,0) == 0.0).
one_test() ->
        ?assert(abiftest:dv(1,5) == 5.0).
two_test() ->
        ?assert(abiftest:dv(2,5) == 2.5).
two_twos_test() ->
        ?assert(abiftest:dv(2,2) == 1.0).
five_test() ->
        ?assert(abiftest:dv(5,5) == 1.0).
```

Fig. 2. A test set for *abiftest*

conjunction this can be triggered by a test in which A is equal to 0, but B is less than 5. This is not covered by any of these test cases.

Mutation testing assumes that all tests pass for the original source file. For the test results produced for each mutant, if any of the tests has failed then the test suite was able to identify the change. This is referred to as *killing* the mutant. If all the tests pass, then the change was not detected and the mutant remains *alive*. The count of *killed* vs *alive* mutants gives a numerical assessment of the fault identification power of the test suite, which meets the requirement for a general quality metric.

Reviewing the specific mutants that remained alive can give much more detailed information about the weaknesses of the test set. That a particular change went unnoticed by the test set implies that that section of the program is not adequately tested. This is the primary reason why the *mu2* framework produces separate mutant files with only one mutation in each file, since this allows clear identification of the specific change that was not detected. This detail about each undetected mutation provides the required identification of specific weaknesses of the test set, and the fact that it is tied directly to the code provides immediate guidance on the areas of the testing to improve.

3 The *mu2* Framework

Overview We have developed the *mu2* framework to automate and simplify the process described in Section 2.2, but also to allow the definition of domain-specific mutation operators in an efficient way.

To support mutation testing the source file of the program is parsed and analysed, and possible mutations identified. Each mutant is produced by applying one mutation operator to one point in the program. This allows the mutation

testing results to identify and characterise specific weaknesses of the test suite in both particular areas of the program and particular styles of fault. To make the mutation testing efficient it is preferable to first identify all the possible mutations, and then select mutations from the list to produce mutants, thus preventing the creation of multiple mutants with identical mutations.

Mutation operators are defined as a triple that includes a name, a function to identify applicable parts of the program, and a function to apply the change. The detailed structure of mutation operators is covered in Section 4. By automating the application of mutation operators and the generation and collation of mutants, *mu2* allows a user to concentrate their efforts on developing innovative and rich mutation operators that reflect the specific faults they want to simulate, and against which they want to evaluate their test suite.

Mutant Generation The *mu2* tool[1] takes as input an Erlang source file and a set of mutation operators. The source file is parsed and all of the possible applications of each of the mutation operators are enumerated. Mutants are then generated by selecting from the possible applications and producing a new Erlang source file with the mutation applied and a header comment added to describe the type and location of the mutation. The result is a folder containing as many mutant files as requested — up to the number of possible mutation applications. The test suite can then be run against each of these modules and the pass or fail status recorded.

The *mu*2 module provides the *generate* function to produce mutants. It can take parameters to specify a particular subset of available operators, or a limited number of mutants, but in its simplest form it takes the source file and an output folder:

```
Eshell V6.0  (abort with ^G)
1> mu2:generate("abiftest.erl","mutants").
Checking applicability of plus_to_minus, 98 more to try...
The current file under checking is:
"abiftest.erl"
Checking applicability of plus_to_mul, 97 more to try...
The current file under checking is:
"abiftest.erl"
[...]
Applying gt_to_lt at {{5,22},{5,26}}...
Renaming to "abiftest_gt_to_le_5_22_5_26"
Writing "mutants/abiftest_gt_to_le_5_22_5_26.erl"...
Applying eq_to_le at {{5,9},{5,14}}...
Renaming to "abiftest_eq_to_le_5_9_5_14"
Writing "mutants/abiftest_eq_to_le_5_9_5_14.erl"...
[...]
```

[1] The *mu2* Erlang mutation testing framework is available at: https://github.com/ramsay-t/mu2

This produces a series of files in the output folder, each names according to the original module name, the mutation operator applied, and the line and character position of the application.

```
-module(abiftest).
-export([dv/2]).

dv(A,B) ->
    if (A == 0) and (B > 4) ->
            B;
       true ->
            B / A
    end.
```

```
-module(abiftest_gt_to_le_5_22_5_26).
-export([dv/2]).

dv(A,B) ->
    if  (A == 0) and (B =< 4) ->
            B;
       true ->
            B / A
    end.
```

Fig. 3. The application of the gt_to_le operator to abiftest.erl

The mutant *abiftest_gt_to_le_5_22_5_26* is shown in Figure 3, next to the original source file. The name represents that it is built from the *abiftest* module by applying the *gt_to_le* operator (replacing a "greater than" operator with a "less than or equals" operator) at line 5, characters 22 through 26.

```
-module(abiftest).
-export([dv/2]).

dv(A,B) ->
    if (A == 0) and (B > 4) ->
            B;
       true ->
            B / A
    end.
```

```
-module(abiftest_eq_to_le_5_9_5_14).
-export([dv/2]).

dv(A,B) ->
    if  (A =< 0) and (B > 4) ->
            B;
       true ->
            B / A
    end.
```

Fig. 4. The application of the eq_to_le operator to abiftest.erl

In a similar fashion, Figure 4 shows the application of the *eq_to_le* operator, replacing an equals operator with a "less than or equals" operator.

Test Set Evaluation The exact process of evaluating the original test set against each mutant may vary between projects if there are substantial requirements for the testing environment. However, the *mu2* framework includes some support functions to evaluate a test set against mutants. The *test* function takes a source folder, module name, mutant folder name, and a test function. It then takes each of the mutants in turn and moves them into the source folder, refactors the mutant name to the original module name, then compiles the module and runs the tests with the mutant in place of the original module.

```
32> Res = mu2:test(".",abiftest,"mutants",fun abiftest_tests:test/0).
Testing "mutants/abiftest_and_to_or_5_9_5_26.erl"
Renaming to "abiftest"
Writing "./abiftest.erl"...
Loading "./abiftest.erl"
abiftest_tests: two_test...*failed*
[...]
[{"abiftest_and_to_or_5_9_5_26.erl",error},
 {"abiftest_and_to_xor_5_9_5_26.erl",error},
 {"abiftest_div_to_minus_6_16_6_20.erl",error},
 {"abiftest_div_to_mul_6_16_6_20.erl",error},
 {"abiftest_div_to_plus_6_16_6_20.erl",error},
 {"abiftest_div_to_rem_6_16_6_20.erl",error},
 {"abiftest_eq_to_ge_5_9_5_14.erl",error},
 {"abiftest_eq_to_gt_5_9_5_14.erl",error},
 {"abiftest_eq_to_le_5_9_5_14.erl",ok},
 {"abiftest_eq_to_lt_5_9_5_14.erl",ok},
 {"abiftest_eq_to_ne_5_9_5_14.erl",error},
 {"abiftest_eq_to_nte_5_9_5_14.erl",error},
 {"abiftest_eq_to_te_5_9_5_14.erl",ok},
 {"abiftest_exchange_if_guard_5_5_7_7.erl",error},
 {"abiftest_exchange_if_pattern_5_5_7_7.erl",error},
 {"abiftest_gt_to_eq_5_22_5_26.erl",ok},
 {"abiftest_gt_to_ge_5_22_5_26.erl",ok},
 {"abiftest_gt_to_le_5_22_5_26.erl",ok},
 {"abiftest_gt_to_lt_5_22_5_26.erl",ok},
 {"abiftest_gt_to_ne_5_22_5_26.erl",ok},
 {"abiftest_gt_to_nte_5_22_5_26.erl",ok},
 {"abiftest_gt_to_te_5_22_5_26.erl",ok},
 {"abiftest_remove_last_if_5_5_7_7.erl",error},
 {"abiftest_swap_if_order_5_5_7_7.erl",ok},
 {"abiftest_true_to_false_6_8_6_11.erl",error}]
```

The final result is a collation of mutant names and the atom *ok* or *error* to indicate the success or failure of the test set. Converting this into a simple killed/alive ratio is simple, but the retention of the mutant names allows the user to trace any mutants that survive and identify the weakness in the test set. In this simple example 11 of the 25 mutants were not killed. Of these only one can be considered semantically equivalent: replacing the == operator with the type-specific =:= is irrelevant, since it would not be meaningful for the function to behave differently with 0.0 than with 0.

However, all of the remaining tests highlight the weakness in the test set — namely that it does not adequately explore the combinations of ways of satisfying and falsifying the condition over A and B. The majority of surviving mutants have modified the condition $B > 4$ on line 5, characters 22 to 26. Although this

condition is exercised, it is only exercised in a limited range of settings (it is only falsified when the other part of the decision is also falsified).

The addition of this test identifies that error in the original code (and, incidentally, achieves full MC/DC coverage of the system as discussed in [16]):

```
div_zero_test() ->
        ?assert(abiftest:dv(0,2) == 2.0).
```

This succinctly demonstrates the way the *mu2* framework meets the test adequacy metric requirements from Section 2.2. The mutation *score* of 11/25 is a general quality metric for the test set, and can be expressed as a percentage with the test set killing only 56% of mutants. This may not be directly comparable to other test adequacy metrics, but gives a similar intuition about the quality of the test suite that a 56% coverage score would do.

That the surviving mutants were predominantly located on line 5, characters 22 to 26 identified not only that decision point as the weakly tested element, but also identified the specific condition that was inadequately tested. Mutation testing is not limited to decision points, as collection of surviving mutants on any other program element (e.g. an output or a calculation) would provide similar evidence that that specific element was inadequately tested. Additionally, the mutants give some guidance on producing new tests, since it highlights some examples that should be distinguished – e.g. if $B > 4$ and $B < 4$ are not distinguished then clearly the system should be tested with values of B both greater and less than 4 (and, perhaps, B equal to 4).

4 Operator Definitions

Practical mutation testing requires that changes be made to the source program's parsed form rather than its source code. Erlang has many libraries in the standard installation that support the parsing of Erlang programs to an abstract syntax tree, but modifications to these trees can be complicated to specify and difficult to understand. The mutation operators that are used in this work can require quite subtle semantic changes that would be particularly complicated to specify in terms of standard syntax tree alterations.

To provide a more succinct and readable interface this work leverages the Wrangler refactoring system [13]. Wrangler is a refactoring system that presents an emacs interface, but it also contains a programatically accessible API. The Wrangler API allows refactorings to be specified in an elegant template format.

In general, *refactoring* is a process that changes a program's source code structure in a consistent way. Common refactorings include: renaming a variable everywhere it is used, extracting blocks of repeated code into a new function, or moving code between levels in a class hierarchy. In order to support such changes, refactoring tools require a rich understanding of the semantics of the target language (e.g. to understand scoping issues when renaming variables). Consequently, the Wrangler refactoring system provides an ideal platform on which to build the mutation operators required for Erlang mutation testing.

The template format of the Wrangler refactoring API uses a series of macros to specify code transformations. The `?RULE` macro defines a rule with three components: a pattern of Erlang code to match, a programatic transformation on that code, and a programatic guard statement to limit the application of the rule. Several different macros define the traversal of the abstract syntax tree; the `?FULL_TD_TP` macro traverses all nodes in the tree. As an example, a function to convert addition to subtraction at a specific program location is shown in Figure 5.

```
plus_to_minus(File, Loc) ->
                ?FULL_TD_TP([?RULE(?T("X@ + Y@"),
                        ?TO_AST("X@ - Y@"),
                        api_refac:start_end_loc(_This@)==Loc)],
                    [File]).
```

Fig. 5. The Wrangler API code to convert addition to subtraction

However, this requires an understanding of the Wrangler system, and it requires the application of multiple Wrangler macros. To speed up the development of mutation operators and allow users to focus on interesting semantic operations the *mu2* framework provides a simplified definition structure. A *mu2* mutation operator is a triple containing a name that will be used to identify the change, a function to identify applicable locations, and a function to alter the code. The *mu2* framework also provides several macros to implement common operations.

```
{plus_to_minus,
?MUTATION_MATCH("X@ + Y@"),
?MUTATION_EXCHANGE("X@ + Y@", "X@ - Y@")}
```

Fig. 6. The *mu2* `plus_to_minus` operator definition

Figure 6 shows the same replacement of addition with subtraction but as a *mu2* operator. The `?MUTATION_MATCH` macro provides a simple way to express a location identifier function that simply matches a template, and the `?MUTATION_EXCHANGE` macro is for mutations that are simple rearrangements or syntax modifications that do not alter the meta-variables.

The significant power of the Wrangler API comes from the ability to perform arbitrary operations on the syntax tree as part of the refactoring operation. The meta-variables in the Wrangler patterns allow the syntax components to be manipulated easily. For example, Figure 7 shows a refactoring to re-order the patterns in a case statement, using the `Pats@@@`, `Guards@@@`, and `Body@@@` meta variables that contain lists of the syntax components for the case statement.

```
{swap_case_order,
      ?MUTATION_RESTRICT("case Expr@ of Pats@@@ when Guards@@@ -> Body@@@ end",
 is_valid_pattern_set(Pats@@@)
),
      ?MUTATION("case Expr@ of Pats@@@ when Guards@@@ -> Body@@@ end",
begin
    A = random:uniform(length(Pats@@@)),
    B = random_not_n(length(Pats@@@), A),
    NewPats@@@ = swap(Pats@@@, A, B),
    NewGuards@@@ = swap(Guards@@@, A, B),
    NewBody@@@ = swap(Body@@@, A, B),
    ?TO_AST("case Expr@ of NewPats@@@ when NewGuards@@@ -> NewBody@@@ end")
end)}
```

Fig. 7. The *mu2* `swap_case_order` operator

The `?MUTATION_RESTRICT` macro creates a location identifier that matches a pattern but also evaluates a boolean function over the metavariables (in this example is simply calls another function to check that the list of patterns is valid). The general `?MUTATION` macro produces a modification function that matches a pattern and then performs an arbitrary function over the meta-variables. This function must return a Wrangler Abstract Syntax Tree, but this is simplified by the `?TO_AST` macro that can build an AST from a template, which can reference newly constructed meta-variables.

5 Evaluation

The *mu2* framework was built as part of the EU funded PROWESS project[2] (Property-based testing of web services). As part of the project the tool was evaluated in an industrial context, and we briefly explain here the evaluation and its results.

5.1 Research Questions

As discussed in Section 2.2, a test adequacy metric must produce both a rigorous assessment and measure of the quality of a test set, and useful feedback to guide a test developer to improve the test set. A simple measure of the *rigour* of an adequacy metric is the number of tests required — assuming that the test set developers don't produce spurious or overly-repetitive tests. The requirement for *useful guidance* is more difficult to measure.

Testing methodologies are often measured using code coverage as a test adequacy metric (e.g. [6]). Several coverage metrics are available for Erlang, including basic line coverage metric provided by the *cover* tool that is included in

[2] http://www.prowessproject.eu/

the standard Erlang library, and MC/DC analysis provided by the *Smother* tool [16]. These have the advantage of requiring less time and effort to apply, since they only require the test suite to be run once, rather than once per mutant. However, [4] showed that — for the imperative languages C and Ada — mutation testing was not only able to provide equivalent levels of quality assurance, but also provide complimentary information and guidance to the test developers.

Consequently, the *mu2* framework is evaluated in comparison to these metrics to demonstrate that it provides similar complimentary benefits in a functional programming language. Specifically we choose the following research questions as the basis for our evaluation:

1. Does mutation testing require more tests than other test adequacy metrics (e.g. coverage) to achieve a maximal score?
2. Do test sets with high mutation scores also achieve high coverage scores, or are they testing different system behaviours?
3. How much longer does it take to perform mutation testing on realistic Erlang modules than testing to maximise coverage?
4. How useful is the feedback provided by mutation testing compared to other metrics?

5.2 Evaluation Results

The PROWESS project partner Interoud Innovation[3] evaluated the *mu2* framework by applying the procedure above. They did not develop any domain-specific operators and used the simple arithmetic and structural operators that are included by default with he framework.

The evaluation itself consisted of an iterative process. The participant developers first measured the coverage and mutation score of their current tests set and then used feedback from the coverage tools and mutation testing tool to improve their test set. The feedback - either uncovered sections or code, or un-killed mutants - should prompt the developers to write new tests. Alternatively, particular execution sequences or mutants may be presented but impossible to cover and can be discarded. The augmented test set can be re-run and new feedback generated. This processes was repeated until there were no uncovered sections of code, no un-killed mutants, until all that remains was identified as impossible, or until the developers judged the remaining items not significant.

This produces a collection of test sets, each produced using feedback from a different metric. The original test set is referred to as T, then the test set developed using *cover* is TC, that produced from *Smother* is TS, and that produced from *mu2* is TM. As well as comparing the improvement made in their own metric, the final test sets were each evaluated using all of the metrics. The numerical results for the metric are presented in tabular form below, followed by a description of each of the test sets.

[3] http://www.interoud.com/

	Original test set (T)	**Cover based test set (TC)**	**Smother based test set (TS)**	**Mu2 based test set (TM)**
Coverage(C)	31.17% (77/247 lines)	69.23% (171/247 lines)	53.44% (132/247 lines)	49.79% (123/247 lines)
Coverage(S)	47.99%	73.16%	69.84%	64.17%
Coverage(M)	16.86% (14/83 killed)	49.39% (41/83 killed)	74.69% (62/83 killed)	80.72% (67/83 killed)

Original test set (*T*)

- The original test set contains 4 QuickCheck properties and 81 Eunit tests.

Cover based test set (*TC*)

- This test set has been developed with the aim of improving line coverage using the Cover tool. It adds 67 Eunit tests to the original test set (4 QuickCheck properties and 148 Eunit tests).
- That this test set achieves a higher percentage MC/DC (*Smother*) score than the test set explicitly designed to achieve a high MC/DC score demonstrates a problem with representing MC/DC scores numerically. Since the developers of TC were aiming to simply execute all lines at least once, this test set executes more lines than TS, resulting in a large number of partially explored decisions. Meanwhile, the developers of TS thoroughly explored the decisions that they considered important, and ignored some that they considered less significant. However, that means that TS leaves a number of decisions completely unexplored, and these contribute multiple MC/DC elements to the percentage since they may contain multiple subcomponents with multiple potential evaluations.

Smother based test set (*TS*)

- This test set adds 64 Eunit tests to the original test set (4 QuickCheck properties and 145 Eunit tests).
- This development took approximately 20 hours, although part of that time was needed to find and fix the bug noted below. This development was also finished before reaching the maximum reachable MC/DC coverage since it was considered more than reasonable coverage.

Mutation based test set (*TM*)

- This test set has been developed with the aim of improving mutation coverage using the *mu2* tool. It adds 15 Eunit tests to the original test set (4 QuickCheck properties and 96 Eunit tests).

– This development took approximately 8 hours and tests were added for almost every mutant. Only 16 of the 83 mutants remained alive:
  - 7 of them were not killed because the mutations were semantically equivalent to the original code (e.g. reordering completely specified case clauses or changing =:= to == when comparing atoms).
  - 7 of them were not killed because the mutation can never be executed.This is the case when the mutation is applied in an internal function with controlled input, so that it is impossible from the tests to cause the input that would make the test fail. In one example the mutant changed an equal comparison by a less than comparison but there is no way of passing a value less than normal into the function parameter, because a previous clause would have match for any other allowed value. This is related to the "impossible" conditions required by Smother, but statically determining the limitations on variable values is a complex problem.
  - 2 of them were not killed because tests to kill them would be too difficult to implement and not useful.

5.3 Developer feedback

The Interoud developers provided some subjective comments about the usefulness of the different strategies used to generate test cases, summarised here:

Cover

– Interoud developers think it may be easier to generate tests focusing on lines of code but it produces worse tests than other metrics.

Smother

– The test set developed for Smother was the most difficult to develop by the Interoud developers. Getting complete coverage in clause based functions is hard because it is likely that some unexpected input (i.e. values of different types or forbidden values, mostly on internal functions) is not covered. Interoud only developed tests using Eunit, but using QuickCheck should have made it easy to check all condition combinations.
– About MC/DC coverage, Interoud developers think it is more useful than line coverage, as it focuses on important lines of code.

mu2

– The mutant based test set is very specific, which makes it more time consuming to apply. Interoud developers created tests for this pilot study that aimed to kill mutants in the most isolated way. Since several mutations are different changes to operators in the same condition, some of these tests kill several mutants.

– Interoud developers think that the *mu2* tool is slow generating mutants, although this was not a problem since mutants were generated only once for all developed test sets.
– Regarding mutation coverage, it was the easiest to develop test for and, although they tend to be too specific, they can be refactored later.
– The default set of mutation operators was only of limited value, but Interoud developers think that the investment of time needed to develop more advanced operators would be worthwhile.

5.4 Evaluation Conclusions

As expected, the test set developed using only line coverage feedback (TC) did not achieve high scores in MC/DC or Mutation assessment. The Smother based test set (TS) and the *mu2* based test set (TM) achieved commendable performance when measured by each other's metric, but there were clearly some areas of difference that represent complementary value — features that are best identified with one metric or the other.

It is significant that although the time taken to generate mutants was mentioned as a weakness of the mutation testing approach, the overall time taken to develop the mutation based test set was considerably shorter. The developers report that the mutants provided the most clear and direct feedback as to what feature was not being executed and how to write a relevant test, although there was some concern that these tests were then too narrowly targeted.

6 Conclusions

This work has produced a framework for applying mutation testing to Erlang programs, and that supports a powerful refactoring based system for defining domain-specific mutation operators. The industrial evaluation of the *mu2* framework demonstrated that it was not only valuable as a test adequacy framework in itself, but that it provided complimentary information to MC/DC analysis of the same system.

Although at the start of this work there was no published mutation testing framework for any functional programming language, the recently released *MuCheck* [12] is a mutation testing environment for Haskell, designed to work with Haskell Quick Check. MuCheck provides conventional mutation operators for list operations and for reordering pattern in pattern matching. The authors make a similar observation in Haskell about the impact of mutations on recursive functions and how this can cause divergent behaviour.

The authors of [9] proposed an approach to automatically generating tests that uses mutation testing to identify areas and types of fault that were of interest. By generating and optimising test sets to kill mutants they expect to create test sets more likely to identify real faults in systems. Because both the mutant generation and test set generation is automated this can be run on a large scale if resources are available. Erlang QuickCheck supports *feature based testing*

that allows test generation to be targeted at specific "features". This could be used in conjunction with *mu*2 mutants to apply this approach to Erlang.

The *mu2* framework has been released as an open-source project on GitHub.

References

1. J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1996.
3. T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with quviq QuickCheck. In M. Feeley and P. W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10. ACM, 2006.
4. R Baker and I Habli. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. *IEEE Trans. Software Eng.*, 39(6):787–805, 2013.
5. R Carlsson and M. Rémond. EUnit: A Lightweight Unit Testing Framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 1–1. ACM, 2006.
6. G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.
7. G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 2014. To appear.
8. G. Fraser and N. Walkinshaw. Behaviourally Adequate Software Testing. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.
9. G. Fraser and A. Zeller. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.
10. R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3:279–290, 1977.
11. M. Harman, Y. Jia, and W. B. Langdon. Strong Higher Order Mutation-based Test Data Generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 212–222. ACM, 2011.
12. D. Le, M. A. Alipour, R. Gopinath, and A. Groce. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 429–432. ACM, 2014.
13. H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical report, University of Kent, 2011.
14. P. Reales Mateo, M. Polo Usaola, and J. Offutt. Mutation at the multi-class and system levels. *Sci. Comput. Program.*, 78(4):364–387, 2013.
15. B. G. Ryder and B. Hailpern, editors. *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. ACM, 2007.
16. R. G. Taylor and J. Derrick. Smother - An MC/DC analysis tool for Erlang. In *Proceedings of the Fourteenth ACM SIGPLAN workshop on Erlang*. ACM, 2015. To Appear.