ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis[*]

Technical Report 2006-519

Jeremy S. Bradbury, James R. Cordy, Juergen Dingel

School of Computing, Queen's University
Kingston, Ontario, Canada
{*bradbury, cordy, dingel*}*@cs.queensu.ca*

October 2006

Abstract

Current mutation analysis tools are primarily used to compare different test suites and are tied to a particular programming language. In this paper we present the *ExMAn* experimental mutation analysis framework – *ExMAn* is automated, general and flexible and allows for the comparison of different quality assurance techniques such as testing, model checking, and static analysis. The goal of *ExMAn* is to allow for automatic mutation analysis that can be reproduced by other researchers. After describing *ExMAn*, we present a scenario of using *ExMAn* to compare testing with static analysis of temporal logic properties. We also provide both the benefits and the current limitations of using our framework.

1 Introduction

Mutation [Ham77, DLS78] has been used in the testing community for over 25 years and is traditionally used to evaluate the effectiveness of test suites. Moreover, mutation provides a comparitive technique for assessing and improving multiple test suites. A number of empirical studies (e.g., [ABL05, DR05]) have relied on using mutation as part of the experimental process.

Although mutation as a comparative technique has been used primarily within the testing community, it does have application in the broader area of quality assurance and bug detection techniques. Our work is based on the idea that mutation can be used to assess testing (e.g., random testing, concurrent testing with tools such as IBM's ConTest), static analysis (e.g., FindBugs, Jlint, PathInspector), model checking (e.g., Java PathFinder, Bandera/Bogor), and dynamic analysis. For example, previously we proposed using mutation to compare sequential testing with property based static analysis using Path Inspector and to compare concurrent testing using ConTest with model checking [BCD05]. The goal of comparing different techniques using mutation is to better understand any complementary relationship that might exist and to use the assessment to design improved hybrid techniques to detect bugs. A combined approach to verifying concurrent Java recently used manual mutants to develop a hybrid analysis approach using code inspection, static analysis (FindBugs, Jlint), dynamic analysis and testing [LDG$^+$04].

We propose a generalized approach to experimental mutation analysis (see Figure 1) in which all of the components and artifacts can be interchanged with other components and artifacts. This generalized

[*]This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

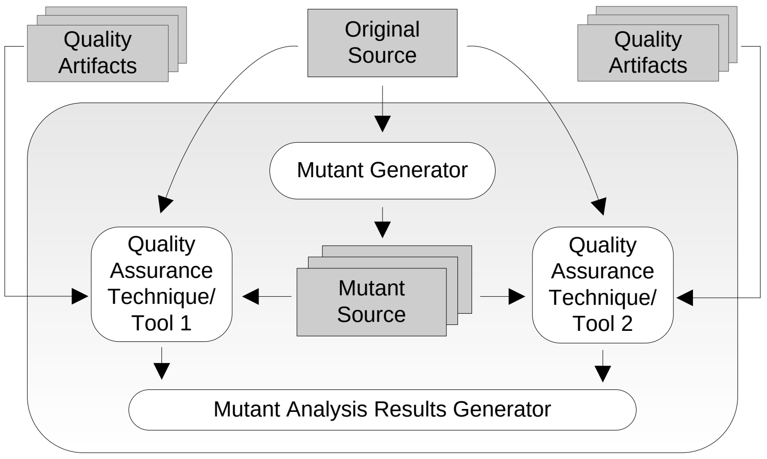

Quality Artifacts
Original Source
Quality Artifacts
Mutant Generator
Quality Assurance Technique/ Tool 1
Mutant Source
Quality Assurance Technique/ Tool 2
Mutant Analysis Results Generator

Figure 1: Generalized Mutation Analysis

approach can be used to compare any number of quality assurance tools that use any kind of quality artifacts. For example, to compare sequential testing with static analysis that uses temporal logic properties. The only restriction is that all tools compared must be able to be applied to the same original source which can be any program language or even an executable modelling language.

Implementing a generalized experimental mutation analysis approach to empirically assess different quality assurance techniques is a challenging problem. A mutation approach that supports the comparison of different quality techniques would have to provide a high degree of automation and customizability. The high degree of automation is required to execute the mutation analysis process and is essential to allow for experimental results to be reproduced. Automation can be achieved through automatically generated scripts to handle the generation of mutants, the mutant analysis, and the generation of results such as mutant score. Customizability is necessary because the approach has to be language and quality artifact independent. On the one hand, language independence means that pluggable mutation generators and compilers are ideal. On the other hand quality artifact independence means the approach should support the comparison of different pluggable quality assurance tools that use artifacts including test cases, assertions, temporal logic properties, and more. In the absence of such a framework, running a wide variety of experiments would mean a considerable duplication of effort.

We have developed the *ExMAn* (EXperimental Mutation ANalysis) framework as a realization of our generalized approach. That is, *ExMAn* is a reusable implementation for building different customized mutation analysis tools for comparing different quality assurance techniques.

In Section 2 we will provide an overview of existing mutation analysis tools that have influenced the design and implementation of *ExMAn*. In Section 3 we will provide a description of *ExMAn*'s architecture as well as the functionality of the *ExMAn* framework. In Section 4 we will provide a scenario of using *ExMAn* for comparing different quality assurance techniques. We will present our conclusions and future work in Section 5.

2 Background

There are several mutation tool including Mothra [DGK$^+$88, DO91], Proteum [DM96], and MuJava [OMK04, MOK05] that our work builds upon. The Mothra tool is a mutation tool for Fortran programs that allows for the application of method level mutation operators (e.g. relational operator replacement). The Proteum tool is a mutation analysis tool for C programs. MuJava is the most recent mutation tool and was designed

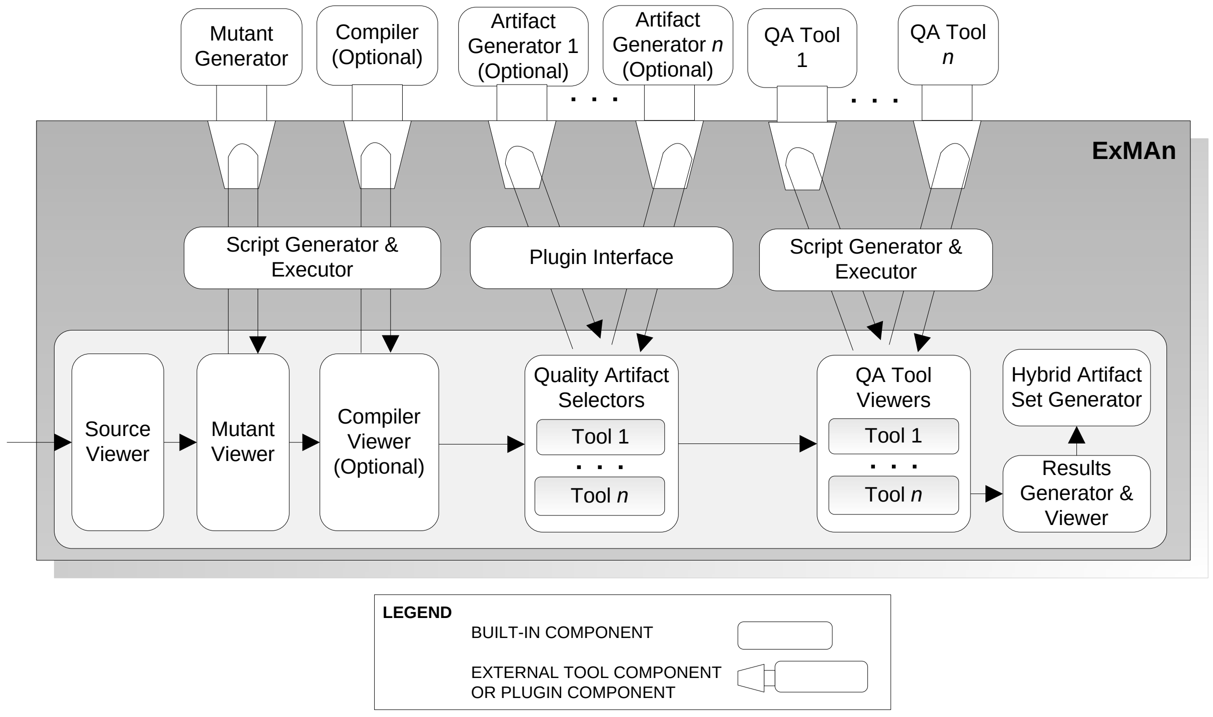

Mutant Generator
Compiler (Optional)
Artifact Generator 1 (Optional)
Artifact Generator *n* (Optional)
QA Tool 1
QA Tool *n*
ExMAn
Script Generator & Executor
Plugin Interface
Script Generator & Executor
Source Viewer
Mutant Viewer
Compiler Viewer (Optional)
Quality Artifact Selectors
Tool 1
Tool *n*
QA Tool Viewers
Tool 1
Tool *n*
Hybrid Artifact Set Generator
Results Generator & Viewer
LEGEND
BUILT-IN COMPONENT
EXTERNAL TOOL COMPONENT OR PLUGIN COMPONENT

Figure 2: *ExMAn* Architecture

The architecture consists of built-in components (appear inside dark grey box) and external tool components and plugin components (appear outside of grey box at top of diagram). The built-in components in the light grey box provide the ExMAn *user interface and allow for control of the external tool components via the Script Generator & Executor. The plugin components are accessed using a plugin interface. Arrows in the diagram represent the typically control flow path between components.*

for use with Java and includes a subset of the method-level operators available in Mothra as well as a set of class mutation operators to handle object oriented issues such as polymorphism and inheritance (e.g. mutate the `public` keyword into `protected`). The difference between *ExMAn* and these tools is that although each is highly automated they were designed to apply mutation analysis to testing. Thus, each is program language dependent and assumes only test cases as quality artifacts. Despite this limitation, all of these tools are excellent for applying mutation analysis to testing and we have learned from their design in building *ExMAn* as a flexible alternative.

3 Overview of *ExMAn*

3.1 Architecture

The *ExMAn* architecture is composed of three kinds of components: built-in components, plugin components, and external tool components. The built-in components are general components that are used in all types of experiments (see Figure 2). We will discuss most of the general components in our description of the *ExMAn* process in Section 3.2. However, we will discuss one important built-in component, the Script Generator & Executor, now. This built-in component provides the interface to the external tool components such as a mutant generator. This component builds and executes scripts when requested by built-in viewer components. Scripts are customized for particular tools based on tool profiles that contain information on the interface of the tool (preferable command line) and a project file that contains information on where input and output are stored. We chose to use a script-based interface for the external tool components because the script interface was more flexible then other interfaces such as a plugin interface and because existing tools are not required to conform to a specific interface.

While the built-in components are general and are used in all mutation analysis experiments, the external components can be replaced, or their usage modified, from one experiment to the next. There are three types of external tool components:

- **Mutant generator.** We can use existing mutant generators such as the Andrews and Zhang C mutant generator tool [AZ03]. We have also designed several custom mutation tools for C and Java using a source transformation langage, TXL [CDMS02].
- **Compiler.** If we are using a testing approach that requires compiled code we can use standard external compilers such as `gcc` or `javac`.
- **Quality Assurance Techniques & Tools.** We can run the mutation analysis on a variety of quality assurance tools including model checkers, static analysis tools, concurrent testing tools, and standard testing techniques.

In addition to the external tool component, there is also one type of plugin component that can be adapted from one experiment to the next:

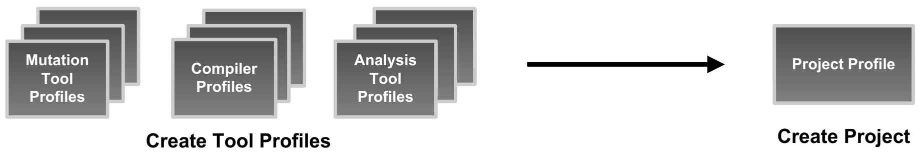
- **Artifact Generator.** We can develop optional customized plugins to generate data for each quality assurance technique in a given experiment. For example, a plugin for testing would produce test cases while a plugin for model checking or static analysis might produce assertions or temporal logic properties.

We have implemented a plugin interface for artifact generators instead of using the script interface because many of the quality assurance tools we are interested in comparing do not have existing artifact generation capabilities. Therefore, we have to create custom generation components instead of using existing external tools. In the future we plan to also provide an alternative script interface for artifact generators to allow us to integrate *ExMAn* with existing test generation tools.

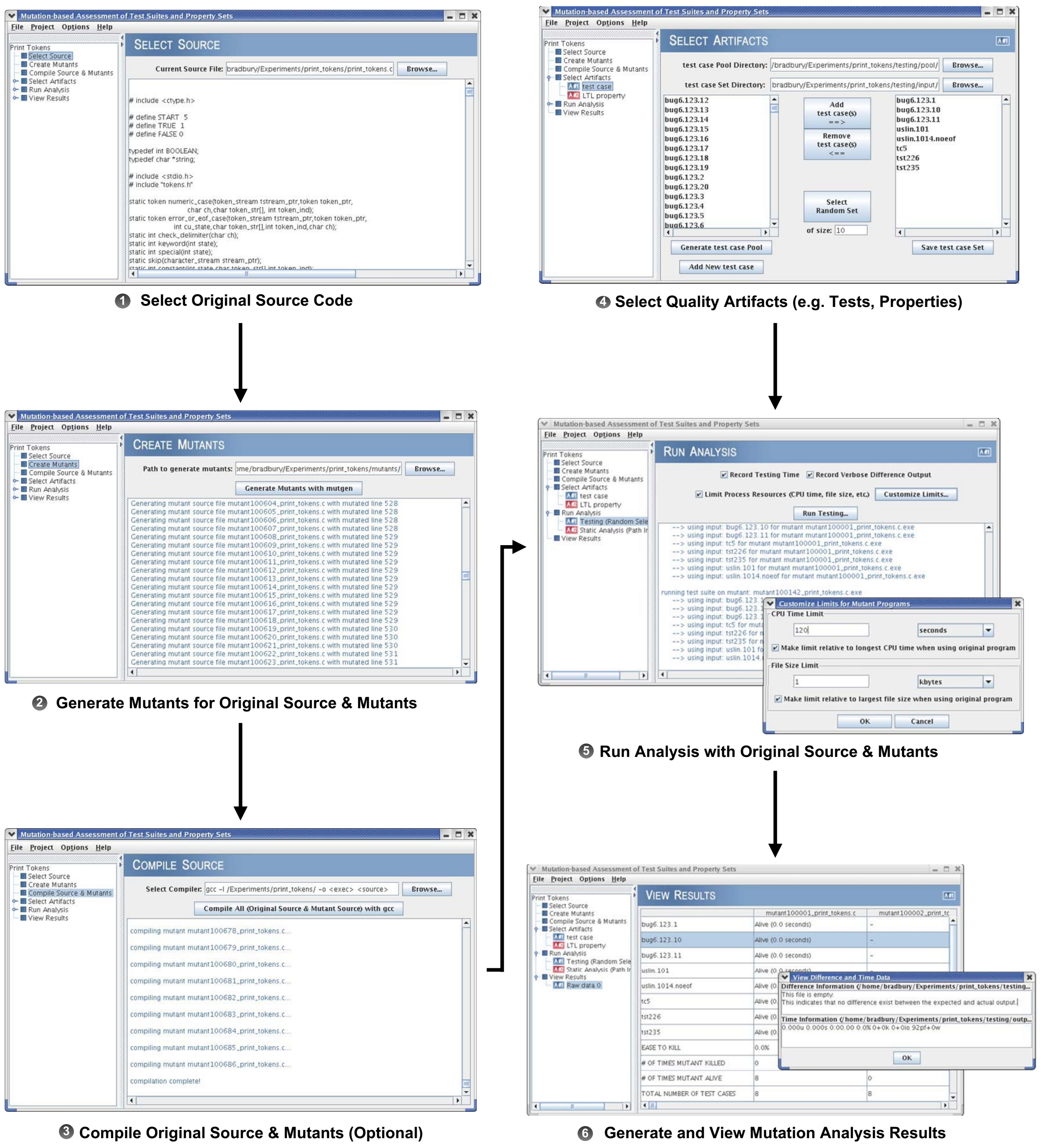
3.2 Process Description

Mutation analysis in *ExMAn* requires a setup phase and an execution phase. The setup phase is required because of the generic and customizable nature of the framework (see Figure 3(a)). Since *ExMAn* is not tied to any language, or analysis tools, profiles have to be created for using *ExMAn* with specific compilers, mutant generators and analysis tools (see Figure 4). A profile contains details on the command-line usage and purpose of the tool. For example to compare concurrent testing using ConTest and model checking using Java PathFinder we would have to ensure that *ExMAn* has defined profiles for a Java compiler, a Java mutant generator (e.g. MuJava) and profiles for executing tests in ConTest as well as model checking with Java PathFinder. *ExMAn* has preinstalled profiles for standard compilers (`gcc`, `javac`), mutant generators, and quality assurance approaches (sequential testing, ConTest, Java PathFinder, Bandera/Bogor, Path Inspector). However these tools have to be installed separately and the profiles might have to be edited to include the correct installation paths.

Once tool profiles have been created, a project for a particular experimental mutation analysis has to be defined (see Figure 5). The project includes information such as the project name and purpose, the compiler (optional), mutant generator, a finite set of quality assurance analysis tools being compared using mutation, and the paths to all input and output artifacts (e.g., test case input and output directories, mutant directory). When reproducing results a previously created project can be used and the setup phase can be bypassed.



(a) Setup phase


Mutation-based Assessment of Test Suites and Property Sets
File Project Options Help
Print Tokens
Select Source
Create Mutants
Compile Source & Mutants
Select Artifacts
test case
LTL property
Run Analysis
Testing (Random Sele
Static Analysis (Path Ir
View Results
Raw data 0
❶ Select Original Source Code
❷ Generate Mutants for Original Source & Mutants
❸ Compile Original Source & Mutants (Optional)
❹ Select Quality Artifacts (e.g. Tests, Properties)
❺ Run Analysis with Original Source & Mutants
❻ Generate and View Mutation Analysis Results

SELECT SOURCE

Current Source File: bradbury/Experiments/print_tokens/print_tokens.c [Browse...]

```
# include <ctype.h>

# define START  5
# define TRUE  1
# define FALSE 0

typedef int BOOLEAN;
typedef char *string;

# include <stdio.h>
# include "tokens.h"

static token numeric_case(token_stream tstream_ptr,token token_ptr,
                          char ch,char token_str[], int token_ind);
static token error_or_eof_case(token_stream tstream_ptr,token token_ptr,
                      int cu_state,char token_str[],int token_ind,char ch);
static int check_delimiter(char ch);
static int keyword(int state);
static int special(int state);
static skip(character_stream stream_ptr);
```

CREATE MUTANTS

Path to generate mutants: ome/bradbury/Experiments/print_tokens/mutants/ [Browse...]

[Generate Mutants with mutgen]

```
Generating mutant source file mutant100604_print_tokens.c with mutated line 528
Generating mutant source file mutant100605_print_tokens.c with mutated line 528
Generating mutant source file mutant100606_print_tokens.c with mutated line 528
Generating mutant source file mutant100607_print_tokens.c with mutated line 528
Generating mutant source file mutant100608_print_tokens.c with mutated line 529
Generating mutant source file mutant100609_print_tokens.c with mutated line 529
Generating mutant source file mutant100610_print_tokens.c with mutated line 529
Generating mutant source file mutant100611_print_tokens.c with mutated line 529
Generating mutant source file mutant100612_print_tokens.c with mutated line 529
Generating mutant source file mutant100613_print_tokens.c with mutated line 529
Generating mutant source file mutant100614_print_tokens.c with mutated line 529
Generating mutant source file mutant100615_print_tokens.c with mutated line 529
Generating mutant source file mutant100616_print_tokens.c with mutated line 529
Generating mutant source file mutant100617_print_tokens.c with mutated line 529
Generating mutant source file mutant100618_print_tokens.c with mutated line 529
Generating mutant source file mutant100619_print_tokens.c with mutated line 530
Generating mutant source file mutant100620_print_tokens.c with mutated line 530
Generating mutant source file mutant100621_print_tokens.c with mutated line 530
Generating mutant source file mutant100622_print_tokens.c with mutated line 531
Generating mutant source file mutant100623_print_tokens.c with mutated line 531
```

COMPILE SOURCE

Select Compiler: gcc -I /Experiments/print_tokens/ -o <exec> <source> [Browse...]

[Compile All (Original Source & Mutant Source) with gcc]

```
compiling mutant mutant100678_print_tokens.c...

compiling mutant mutant100679_print_tokens.c...

compiling mutant mutant100680_print_tokens.c...

compiling mutant mutant100681_print_tokens.c...

compiling mutant mutant100682_print_tokens.c...

compiling mutant mutant100683_print_tokens.c...

compiling mutant mutant100684_print_tokens.c...

compiling mutant mutant100685_print_tokens.c...

compiling mutant mutant100686_print_tokens.c...

compilation complete!
```

SELECT ARTIFACTS

test case Pool Directory: /bradbury/Experiments/print_tokens/testing/pool/ [Browse...]

test case Set Directory: bradbury/Experiments/print_tokens/testing/input/ [Browse...]

bug6.123.12	[Add test case(s) ==>]	bug6.123.1
bug6.123.13	[Remove test case(s) <==]	bug6.123.10
bug6.123.14		bug6.123.11
bug6.123.15		uslin.101
bug6.123.16		uslin.1014.noeof
bug6.123.17		tc5
bug6.123.18		tst226
bug6.123.19		tst235
bug6.123.2		
bug6.123.20		
bug6.123.3	[Select Random Set]	
bug6.123.4	of size: 10	
bug6.123.5		
bug6.123.6		

[Generate test case Pool] [Add New test case] [Save test case Set]

RUN ANALYSIS

[✓] Record Testing Time [✓] Record Verbose Difference Output

[✓] Limit Process Resources (CPU time, file size, etc.) [Customize Limits...]

[Run Testing...]

```
   --> using input: bug6.123.10 for mutant mutant100001_print_tokens.c.exe
   --> using input: bug6.123.11 for mutant mutant100001_print_tokens.c.exe
   --> using input: tc5 for mutant mutant100001_print_tokens.c.exe
   --> using input: tst226 for mutant mutant100001_print_tokens.c.exe
   --> using input: tst235 for mutant mutant100001_print_tokens.c.exe
   --> using input: uslin.101 for mutant mutant100001_print_tokens.c.exe
   --> using input: uslin.1014.noeof for mutant mutant100001_print_tokens.c.exe

running test suite on mutant: mutant100142_print_tokens.c.exe
   --> using input: bug6.123.1
   --> using input: bug6.123.1
   --> using input: bug6.123.1
   --> using input: tc5 for muta
   --> using input: tst226 for m
   --> using input: tst235 for m
   --> using input: uslin.101 fo
   --> using input: uslin.1014.
```

Customize Limits for Mutant Programs

CPU Time Limit: 120 seconds

[✓] Make limit relative to longest CPU time when using original program

File Size Limit: 1 kbytes

[✓] Make limit relative to largest file size when using original program

[OK] [Cancel]

VIEW RESULTS

	mutant100001_print_tokens.c	mutant100002_print_tc
bug6.123.1	Alive (0.0 seconds)	-
bug6.123.10	Alive (0.0 seconds)	-
bug6.123.11	Alive (0.0 seconds)	-
uslin.101	Alive (0.0 seconds)	-
uslin.1014.noeof	Alive (0.	
tc5	Alive (0.	
tst226	Alive (0.	
tst235	Alive (0.	
EASE TO KILL	0.0%	
# OF TIMES MUTANT KILLED	0	
# OF TIMES MUTANT ALIVE	8	0
TOTAL NUMBER OF TEST CASES	8	8

View Difference and Time Data

Difference Information (/home/bradbury/Experiments/print_tokens/testing...

This file is empty.
This indicates that no difference exist between the expected and actual output.

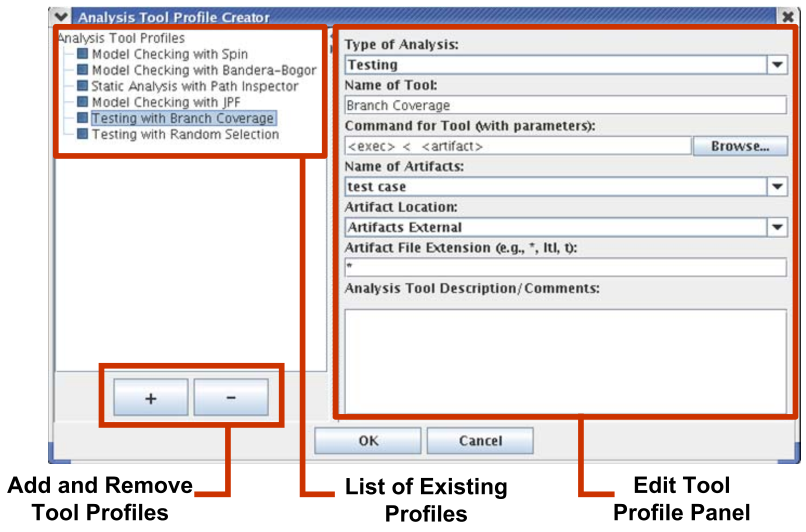
Time Information (/home/bradbury/Experiments/print_tokens/testing/outp...

```
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 92pf+0w
```

[OK]

(b) Execution phase

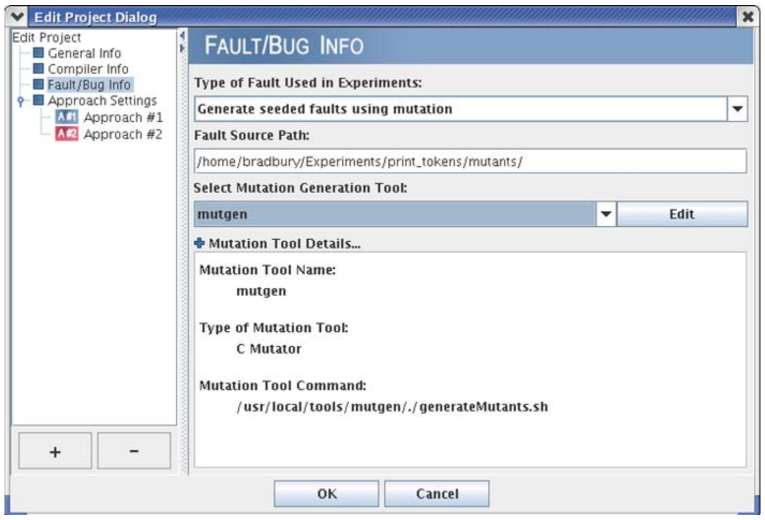
Figure 3: *ExMAn* Process


Analysis Tool Profile Creator
Add and Remove Tool Profiles
List of Existing Profiles
Edit Tool Profile Panel
+
-
OK
Cancel
Browse...

Analysis Tool Profiles
- Model Checking with Spin
- Model Checking with Bandera-Bogor
- Static Analysis with Path Inspector
- Model Checking with JPF
- Testing with Branch Coverage
- Testing with Random Selection

Type of Analysis: Testing
Name of Tool: Branch Coverage
Command for Tool (with parameters): <exec> < <artifact>
Name of Artifacts: test case
Artifact Location: Artifacts External
Artifact File Extension (e.g., *, ltl, t): *
Analysis Tool Description/Comments:

Figure 4: *ExMAn* Tool Profile Creator Dialog



Edit Project
- General Info
- Compiler Info
- Fault/Bug Info
- Approach Settings
  - Approach #1
  - Approach #2

Type of Fault Used in Experiments: Generate seeded faults using mutation
Fault Source Path: /home/bradbury/Experiments/print_tokens/mutants/
Select Mutation Generation Tool: mutgen
Mutation Tool Details...
Mutation Tool Name: mutgen
Type of Mutation Tool: C Mutator
Mutation Tool Command: /usr/local/tools/mutgen/./generateMutants.sh

Figure 5: *ExMAn* Create/Edit Project Dialog

The execution phase occurs once *ExMAn* has been customized for a particular experimental mutation analysis. The execution phase consists of the following steps (see Figure 3(b)):

1. *Original Source Code Selection:* select the program or model source to be used in the mutation analysis. A generic language-independent Source Viewer displays the source but does not do any language specific pre-processing to ensure the source is syntactically correct.

2. *Mutant generation:* the mutant generator specified in the project is used to generate a set of mutants for the original source code. The Mutant Viewer reports the progress of the mutant generation.

3. *Compile Original Source Code & Mutants:* an optional step that occurs only if at least one of the quality assurance tools involves dynamic analysis or testing. The progress of the compilation is reported in the Compile Viewer.

4. *Select Quality Artifacts:* for each quality assurance analysis tool being analyzed using mutation a set of quality artifacts is selected. For example, with model checking a set of assertions can be selected from an assertion pool. The assertion pool can be generated by an optional Artifact Generator plugin or we can use an existing assertion pool. The selection of quality artifacts can be conducted randomly or by hand using a Quality Artifact Selector & Viewer. For example we could randomly select 20 assertions from an assertion pool or select them by hand. Each quality artifact can also be viewed in a dialog interface.

5. *Run Analysis with Original Source Code & Mutants:* Quality Analysis Tool Viewers call automatically generated scripts which allow all of the quality assurance tools to be run automatically. For each tool's set of quality artifacts, we first evaluate each artifact using the original source to determine the expected outputs. Next we evaluate the artifacts for all of the mutant versions of the original program. During this step all of the tool analysis results and analysis execution times of each artifact with each program version are recorded and the progress is reported. Quality Analysis Tool Viewers also provide an interface to customize the running of the analysis by placing limits on the size of output and the amount of CPU time. For example, a mutant might cause the original program to go into an infinite loop and never terminate which would be a problem if we are evaluating a test suite. Fortunately, the user can account for this by placing relative or absolute limits on the resources used by the mutant programs. If relative limits are used then the resources used by the original program are recorded and the resources used by each mutant are monitored and the mutant is terminated once it exceeds a relative threshold (e.g. 60 seconds of CPU time more then the original program).

6. *Collection and Display of Results:* results using all of the quality assurance tools are displayed in tabular form in the Results Generator & Viewer. The data presented includes the quality artifact vs. mutant raw data, the mutant score and analysis time for each quality artifact and the ease to kill each mutant (i.e. the number of quality artifacts that kill each mutant). We also can generate hybrid sets of quality artifacts from all quality assurance tools that have undergone mutation analysis using the Hybrid Artifact Set Generator. For instance, if different artifacts are used with different tools we report the combined set of quality artifacts that will achieve the highest mutant score. Additionally, we can generate the hybrid set of artifacts that achieve a certain mutant score (e.g. 95%) and has the lowest execution cost or smallest set of quality artifacts.

4 *ExMAn* in Practice

We will now outline a scenario that demonstrate *ExMAn*'s flexibility and the novel application of mutation analysis that is possible using our framework. The scenario does not provide a statistical comparison of quality assurance techniques. Instead, the scenario demonstrates customizing *ExMAn* for experimental mutation analysis research.

Consider a scenario where we compare sequential testing and static analysis. In this scenario we can use the following external components and plugins with *ExMAn*:

- **Mutant generator.** The Andrews and Zhang C mutant generator tool [AZ03].
- **Compiler.** `gcc`.
- **Quality Assurance Techniques & Tools.**
  - Technique 1: Sequential testing.
  - Technique 2: Static Analysis using Path Inspector (a tool that allows for the analysis of temporal logic properties).

- **Artifact Generator.** We could compare a test suite selected randomly from an already existing test pool with a set of properties that are selected randomly from a generated pool of temporal properties. Our property generator plugin for Path Inspector first extracts possible property variables from the program and then composes variables using temporal logic property patterns.

Using *ExMAn* with the above customization we could determine the mutant score of each analysis technique and produce the hybrid set that has the highest mutant score. The hybrid set may contain both tests and properties or could potential contain only tests or only properties.

We could also reconfigure *ExMAn* to compare analysis using hand created properties with Path Inspector versus properties created using our property generator plugin.

5 Conclusion

ExMAn is a generic and flexible framework that allows for the automatic comparison of different quality assurance techniques and the development of hybrid quality assurance approaches. The flexibility of *ExMAn* occurs because of the separation of the built-in components that can be used in any mutation analysis from the external tool components that place restrictions on the mutation analysis. By using a script invocation interface to access the mutant generator, compiler and quality assurance techniques under analysis we allow them to be easily interchanged with no modifications to the original tools. We have demonstrated the customization of *ExMAn* using one example and are currently planning empirical assessments of testing, static analysis, and formal analysis.

Although *ExMAn* is a generalized and customizable way to conduct mutation analysis it does have limitations and we have identified several areas of future work:

- Add some facility to semi-automatically or automatically identify equivalent mutants.
- Add ability to automatically specify patterns for the creation of mutation operators.
- Expand the artifact selection to allow for the selection of multiple quality artifact sets for each type and thus allow for statistical analysis.

We are interested in improving the functionality and flexibility of the *ExMAn* framework and hope to address the above limitations in the near future.

6 Availability

We are currently using *ExMAn* to conduct several empirical studies regarding the bug detection abilities of testing vs. property-based analysis for both sequential and concurrent systems. Once we have completed these experiments and evaluated the usability and effectiveness of *ExMAn* we plan to publicly release *ExMAn* in January 2007.

References

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of* 27th *International Conference on Software Engineering (ICSE 2005)*, pages 402–411, 2005.

[AZ03] James H. Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29(7):634–648, 2003.

[BCD05] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. An empirical framework for comparing effectiveness of testing and property-based formal analysis. In *Proc. of* 6th *Int. ACM SIGPLAN-SIGSOFT Work. on Program Analysis for Software Tools and Engineering (PASTE 2005)*, Sept. 2005.

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *J. of Information and Software Technology*, 44(13):827–837, 2002.

[DGK$^+$88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. An extended overview of the Mothra software testing environment. In *Proc. of the* 2nd *Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Jul. 1988.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints for test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[DM96] M. Delamaro and J. Maldonado. Proteum–a tool for the assessment of test adequacy for c programs. In *Conf. on Performability in Computing Sys. (PCS 96)*, pages 79–95, Jul. 1996.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[DR05] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proc. of the* 21st *IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 411–420, Washington, DC, USA, 2005. IEEE Computer Society.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Soft. Eng.*, 3(4), Jul. 1977.

[LDG$^+$04] Brad Long, Roger Duke, Doug Goldson, Paul A. Strooper, and Luke Wildman. Mutation-based exploration of a method for verifying concurrent Java components. In *Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, Jun. 2005.

[OMK04] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for Java. In *Proc. of the Workshop on Empirical Research in Software Testing (WERST'2004)*. SIGSOFT Software Engineering Notes, 29(5):1–4, ACM Press, 2004.