Automated Unit Testing Tool for .Net Framework

Doaa Sami
Computer Science,
Misr University
Cairo, Egypt

Hala Abdel-Galil,Ph.D
Computer Science,
Helwan University
Cairo, Egypt.

Prof.Mostafa Sami
Computer Science
Helwan University
Cairo, Egypt.

ABSTRACT
Developers use unit testing to improve the quality of software systems. Current development tools for unit testing help with automating test execution, with reporting results, and with generating test stubs. However, they offer no aid for designing tests aimed specifically at exercising the effects of changes to a program. This paper describes a unit testing tool that makes writing unit tests Easier and more efficient by introducing an open source unit testing tool for the .NET Framework. Unit testing is tightly associated with test-driven development (TDD), refactoring, and other practices from agile software development approaches such as Extreme Programming or Scrum [19, 20]. The tool provides developers with the Standard unit testing features such as test, fixture, setup, teardown, ignores, expected exception, etc. The tool has an easy graphical user interface to facilitate to the user the testing process. The tool also has a lot of advanced features like the Recipes which make the user able to combine several test assemblies into one test suite plus the Search capabilities across tests, output, and statistics and also generates Statistics per test to create performance base line and grouping tests by categories for execution and works with any .Net language.

Keywords
Unit testing, system testing, test framework.

1. INTRODUCTION
Software quality assurance is in dire need of substantial progress because Software programs continue to evolve throughout their Lifetime. Maintaining such evolving programs is one of the most expensive activities in the process of software development.Software testing is resource-hungry, time-consuming, labor-intensive, and is the most widespread way of uncovering faults in software. Despite massive investments in quality assurance, serious code defects are routinely discovered after software has been released, and fixing them at so late a stage carries substantial cost.

In the last few years, unit testing, which targets small, self-contained sections of code, has been widely adopted.

Unfortunately, classical unit testing has several shortcomings, as confirmed by a software developer with several years of experience in industry:

Functions that change the state of a complex component are difficult to test individually. Sometimes, it is impossible to tell if a certain internal state is correct [15, 16].

Some tests require a complex context that is difficult to set up.

Other tests rely heavily on other modules that are still under development.

All today software unit testing concern only one programming language but today large projects can combine many programming languages under one framework like visual studio.net.

While most developers agree on the advantages of having a solid test suite with good code coverage, most also admit the difficulty of developing such a test suite [18].

In this paper, presenting a unit testing tool, which is aware of the developer's edit in a program, and thus can guide him in writing those unit tests that effectively exercise all changed parts of a program and their effects on program behavior and give the user a lot of traditional and new facilities that will help him to test in an efficient way plus it can test a software written in any language under .NET Framework.

2. THE UNIT TESTING TOOL
This is a cost-effective and comprehensive tool used for automatic testing. This is a better alternative to conventional testing tools because it tests applications from a user's perspective, using standard programming techniques and common languages such as C# and VB.net. It does not require the tester to learn a scripting language, because it is written in pure .net code. Tester can use any one of the .NET languages. The tool is based on the JCrasher algorithm which works by building a graph of input generation methods and traversing this graph in order to create test-cases which is then executed in a runtime environment, which is a very good way to specify a set of random values to be provided for an individual parameter of a parameterized test method. It is a pure .net API, which is very different from other tools which sit on an API. Future plans for this tool involve creating an open and documented interface for the users to write their own plug-ins, which provides the maximum of object recognition for their own applications.

The tool has the following major features:

- Standard unit testing features such as test, fixture, setup, teardown, ignore, expected exception, etc.
- The testing process based on both different attributes and rich set of assertions
- The Unit Testing Tool support parameterized tests with as many parameters as needed.
- Built on the .NET Framework.
- Offers a flexible and standard test automation interface
- The test automation modules can be created as simple executable builds, with a standard .NET compiler.
- The unit testing tool automation library (API) is based on .NET, therefore, allowing tester to integrate it into existing test environments and to combine existing automation tasks with the tool.

- Provides the ability to do test automation in tester's own environment
- Uses standard and modern programming techniques
- Allows testers with less programming knowledge to create professional test modules
- Targets to get everything automated
- User interface allows for managing test cases and configurations.
- Categories to group tests for execution.

3. THE DESIGN OF THE TOOL

The Unit testing Tool consists of several components as shown in Figure [1]:

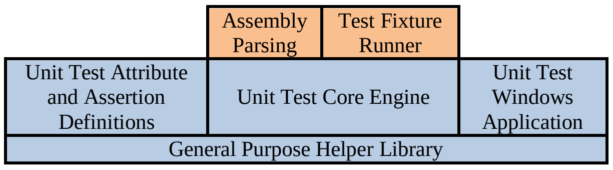
Assembly Parsing
Test Fixture Runner
Unit Test Attribute and Assertion Definitions
Unit Test Core Engine
Unit Test Windows Application
General Purpose Helper Library

Fig.1: over view of The Unit Testing Tool

- General Purpose Helper Library
- **Unit Test Attribute and Assertion Definitions:** This assembly consists of the necessary definitions for implementing a unit test class. The attributes that are associated with a unit test class and its methods must are defined. Similarly, the assertions that the unit test methods can perform are defined. This assembly is the only assembly that needs to be referenced by a unit test assembly.
- **Unit Test Core Engine:** This component consists of two pieces:
  - **general assembly parsing functions:** which extract out the classes and methods in an assembly and their attributes
  - **unit test automation:** which consists of creating test fixtures, managing the classes and methods in a test fixture, and running the tests
- **Unit Test Windows Application:** The Window Forms application consists of three sections:
  - a tree view showing all the unit test classes, their methods, and the specific test results
  - a progress bar providing the user with feedback as to the progress of the test cases
  - a summary of test results, showing the count of passed, ignored, and failed tests
- **The Unit Test Core Engine:** A high level block diagram of the test apparatus can be illustrated as shown in Figure [2].

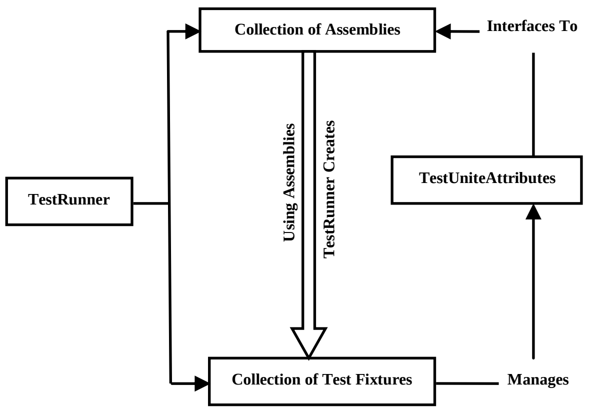

Fig.2: the test apparatus

4. THE TOOL'S TESTING TECHNIQUE

The unit testing tool runs upon certain events in the test cycle by decorating fixture classes and methods with the appropriate provided attributes as in figure [3]. The tool consists of setup code that prepares the test input, then executes some test code on the unit under test, and finally assesses the observed behavior.
Action Attributes allow the user to create custom attributes to encapsulate specific actions for use before or after any test is run.

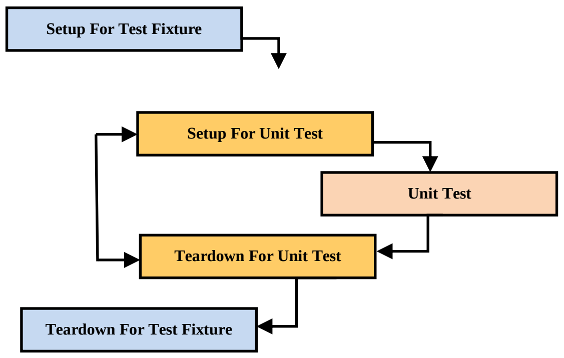

Fig.3: The unit testing tool cycle.

The unit testing tool also may use the random attribute to specify a set of random values to be provided for an individual parameter of the parameterized test method [2, 4]. By using the JCrasher algorithm which works by building a graph of input generation methods and traversing this graph in order to create test-cases which is then executed in a runtime environment [3]. The unit testing tool creates test cases from all possible combinations of the datapoints provided on parameters. The recent success of parameterized unit testing is based on the ability to cover diverse behavior with a single case: The same test code executed with different inputs can show different behavior.
Retrofitting of unit tests [9] is an approach where existing unit tests are converted to parameterized unit tests, by identifying inputs and converting them to parameters, and by generalizing assertions to oracles that hold independently of the input.
An additional advantage of parameterized unit tests is that they are easier to understand: There is less test code, and this code is also independent of the concrete inputs. Test factoring describes a related technique where an existing test case is converted to improve aspects such as readability or execution

speed. For example, unrelated objects can be replaced with mock objects [10], and minimization [11, 12] reduces the length of automatically generated test cases.

During the coding and testing process, programmer must make sure that his tests do not leave important parts of the code untested [5]. The unit testing tool helps write tests, but selecting a complete test set is up to the individual programmer [6]. The unit testing tool used the path coverage metric for test coverage by using the Mutation criteria to ensure generation of all paths needed from the unit testing and to aid the tester to identify possible oracles [1, 8].

The all-paths criterion, which requires at least one test-case per feasible path of the function under test, is recognized as offering a high level of software reliability.

5. THE MUTATION CRITERIA

The coverage criteria are used to guide test generation and to reduce the number of tests that are generated [7]. The mutation coverage is based on syntactic faults. For code-based unit level testing, small changes are introduced into the program (mutants) and tests are required to cause each mutant to result in incorrect output [14].

Choosing the mutation criteria among four famous different criteria because it’s Effectiveness as it will help the tester find more faults .Effectiveness is approximated by the number of faults detected.

The tool’s algorithm combines the pre- and postconditions with the test case into a parameterized unit tests, containing all information in a single package. In detail:

Test parameterization: By converting concrete method sequences into parameterized test cases, reducing the number of statements the developer has to analyze.

Postconditions: By mutating the tested class then identifies the relevant aspects of the postconditions, and suggests oracles that are effective at finding defects.

Preconditions: By mutating the test inputs then identify the relevant aspects of the preconditions, and filter out overly specific postconditions.

Iterative refinement: By using a search-based approach to iteratively derive new test inputs that aim at removing further preconditions, thus simplifying the test case.

Efficiency is approximated by the number of tests needed to satisfy the criteria. To generalize a method sequence to a parameterized method sequence, we assume one dedicated class as UUT. All method calls on this UUT are part of the test code, and objects created by calls to the UUT are also part of the test; all remaining calls are considered to be inputs. The length of the test case is the number of calls n. A call can be a call to a constructor, a method, or an assignment of a primitive value or object member to a test object. The mutation criteria algorithm used:

Algorithm1 Parameterize Test Case

Require: Call Sequence $M = (m_1, \dots, m_n)$
Require: Class under Test C
Ensure: Parameterized Unit Test $P = (\boldsymbol{I}, T, Pre, Post)$

```
1:  procedure PARAMETERIZE($M, C$)
2:      $G \leftarrow (V, E)$
3:      $S \leftarrow \{\}$
4:      for all $m \in M$ do
5:          $v \leftarrow value(m)$
6:          $V \leftarrow V \cup \{v\}$
7:          for all $v^r \in params(m)$ do
8:              $G \leftarrow G \cup \{(v^r, v)\}$
9:          end for
10:         if $m$ is a call of $C$ then
11:             $S \leftarrow S \cup \{v\}$
12:             $T \leftarrow T.m$
13:         end if
14:     end for
15:     for all $v \in S$ do
16:         for all $(v^r, v) \in V$ do
17:             if $v^r \notin S$ then
18:                 $\boldsymbol{I} \leftarrow \boldsymbol{I} \cup \{$ New parameter with type of $v\}$
19:                 $p \leftarrow$ Backwards slice of $v^r$
20:                 $Pre \leftarrow Pre \cup \{$ Extract conditions for $p\}$
21:             end if
22:         end for
23:     end for
24:     $Post \leftarrow \{$ Extract conditions for each value in $T\}$
25:     return $P$
26: end procedure
```

The algorithm appears how a call sequence is converted to a parameterized unit test. First, generating a graph in which there is a vertex for every value, and edges between values if the call producing a value has dependencies on other values. By separating values that are test code from those values that are setup code, one can easily determine inputs: For each test vertex there is one input for every incoming edge that does not come from another test vertex. The graph easily lets us derive method sequences to construct each of the parameters; the calls that are part of the test and not part of an input are added to T. If the same input value is used by different calls, then each of the uses results in a distinct input.

Algorithm 2 illustrates how a set of postconditions is reduced to the relevant subset: The test case t is executed against every single mutant, and a postcondition only qualifies as relevant oracle, if there exists at least one mutant for which the assertion fails [13].

Algorithm2 Determine effective postconditions

Require: Call Sequence $M = (m_1, \dots, m_n)$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Require: Set of Postconditions $Post$
Ensure: Reduced Set of Postconditions $Post^r$

```
1:  procedure FINDEFFECTIVE($M, C, \mathcal{M}, Post$)
2:      $Post^r \leftarrow \{\}$
3:      for all $m \in \mathcal{M}$ do
4:          $S \leftarrow$ state after executing $M$ on $m$
5:          for all $p \in Post$ do
6:              if $p$ evaluates to false in $S$ then
7:                  $Post^r \leftarrow Post^r \cup \{p\}$
8:              end if
9:          end for
10:     end for
11:     return $Post^r$
12: end procedure
```

Algorithm 3 illustrates how a set of postconditions can be gradually reduced to retain only general postconditions that hold for more than one input [13].

Algorithm3 Determine robust postconditions

Require: PUT $P = (\boldsymbol{I}, T, Pre, Post)$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Ensure: Reduced Set of Postconditions $Post$

```
1:  procedure FINDEFFECTIVE($M, C, \mathcal{M}, Post$)
2:      $Post^r \leftarrow \{\}$
```

```
 3:   for all $m \in \mathcal{M}$ do
 4:      $S \leftarrow$ state after executing $M$ on $m$
 5:      for all $p \in Post$ do
 6:         if $p$ evaluates to false in $S$ then
 7:            $Post^r \leftarrow Post^r \cup \{p\}$
 8:         end if
 9:      end for
10:   end for
11:   return $Post^r$
12: end procedure
```

Algorithm 4 illustrates how to reduce the preconditions of a given parameterized unit testing.

Algorithm4 Generate a Parameterized Unit Test

Require: Call Sequence $M = (m_1, \ldots, m_n)$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Ensure: Parameterized Test Case $P = (\mathcal{I}, T, Pre, Post)$

```
 1: procedure GENERALIZE($M, C, \mathcal{M}$)
 2:    $P \leftarrow$ PARAMETERIZE($\mathcal{M}, C$)
 3:    $Pre^r \leftarrow$ minimized observations on inputs
 4:    $Post \leftarrow$ observations on $C$ after test execution
 5:    $Post \leftarrow$ FINDEFFECTIVE($M, C, \mathcal{M}, Post$)
 6:    while $Pre^r$ is not empty do
 7:       $p \leftarrow$ remove one element from $Pre^r$
 8:       $\mathcal{I} \leftarrow$ generate input that satisfies $\neg p \wedge \bigvee Pre^r \wedge \bigvee Pre.$
 9:       if test generation succeeds then
10:          $Post^r \leftarrow$ Execute test with new input $\mathcal{I}$
11:          $M \leftarrow$ concrete method sequence of $\mathcal{I}$ and $T$
12:          $Post^r \leftarrow$ FINDEFFECTIVE($M, C, \mathcal{M}, Post^r$)
13:          if $Post^r \cap Post$ detects all mutants then
14:             $Post \leftarrow Post^r \cap Post$
15:             $Pre^r \leftarrow Pre' \cup$ conditions subsumed by $p$
16:          else
17:             $Pre \leftarrow Pre \cup \{p\}$
18:          end if
19:       end if
20:    end while
21:    return $P$
22: end procedure
```

6. CONCLUSIONS

The Unit Testing Tool is software that can help developers to write more effective tests. Moreover, full change coverage is not just an achievable goal but also seems to reduce the likelihood of introducing faults to the program. This correlation and a change-centric test development approach are major targets of the current evaluation.

The unit testing tool technique for "test" generation actually does not produce tests—it produces sequences of method calls. While it is good at covering code, their effectiveness relies on good run-time checks in the code or the run-time system.

Finally, The Unit Testing Tool is a good example for the potential of change-aware tools. It demonstrates how even complex development activities can be supported by tools when they are aware of what a developer has done to the code.

7. REFERENCES

[1] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis, pages 147–158. ACM, 2010.

[2] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pages 815–816, New York, NY, USA, 2007. ACM.

[3] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. Software Practice and Experience, 34(11):1025–1050, 2004.

[4] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In RT '06: Proceedings of the 1st International Workshop on Random Testing, pages 36–45, New York, NY, USA, 2006. ACM.

[5] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In EDCC 2005: Proceedings ot the 5th European Dependable Computing Conference, volume 3463 of LNCS, pages 281–292. Springer, 2005.

[6] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit etesting engine for C. In ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 263–272, New York, NY, USA, 2005. ACM.

[7] N. Tillmann and J. N. de Halleux. Pex— white box test generation for .NET. In TAP 2008: International Conference on Tests and Proofs, volume 4966 of LNCS, pages 134 –253. Springer, 2008.

[8] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In TAIC-PART '09: Proceedings of Testing: Academic & Industrial Conference – Practice and Research Techniques, pages 95–104, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[9] S. Thummalapenta, M. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2011), 2011.

[10] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pages 114–123, New York, NY, USA, 2005. ACM.

[11] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In TAIC-PART '09: Proceedings of Testing: Academic & Industrial Conference - Practice and Research Techniques, pages 95–104, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[12] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, pages 267 276, Washington, DC, USA, 2005. IEEE Computer Society.

[13] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis, pages 147–158. ACM, 2010.

[14] Mike Papadakis, Nicos Malevris, Maria Kallia,Towards automating the generation of mutation tests, AST '10: Proceedings of the 5th Workshop on Automation of Software Test, May 2010.

[15] Whittaker, James A. Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, Addison-Wesley Professional, 2009, 978-0321636416.

[16] Kaner, Cem et al. Lessons Learned in Software Testing. Wiley, 2001. 978-0471081128.

[17] Desikan, S. and Ramesh, G. Software Testing: Principles and Practices, Pearson Education, 2006.

[18] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In Workshop on Large Scale Distributed Systems and Middleware, 2009.

[19] Hendrickson, E. "Agile Testing, Nine Principles and Six Concrete Practices for Testing onAgileTeams"http://testobsessed.com/wordpress/wpcontent/uploads/2008/08/AgileTestingOverview.pdf 2008, accessed 07/09/2010.

[20] K. Beck. Test Driven Development: By Example. Addison-Wesley, 2003.