Unit Test Code Generator for Lua Programming Language

Junno Tantra Pratama Wibowo, Bayu Hendradjaya, Yani Widyani
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
junnotantra@gmail.com, bayu@informatika.org, yani@informatika.org

Abstract— **Software testing is an important step in the software development lifecycle. One of the main process that take lots of time is developing the test code. We propose an automatic unit test code generation to speed up the process and helps avoiding repetition. We develop the unit test code generator using Lua programming language.**

Lua is a fast, lightweight, embeddable scripting language. It has been used in many industrial applications with focuses on embedded systems and games. Unlike other popular scripting language like JavaScript, Python, and Ruby, Lua does not have any unit test generator developed to help its software testing process.

The final product, Lua unit test generator (LUTG), integrated to one of the most popular Lua IDE, ZeroBrane Studio, as a plugin to seamlessly connect the coding and testing process. The code generator can generate unit test code, save test cases data on Lua and XML file format, and generate the test data automatically using search-based technique, genetic algorithm, to achieve full branch coverage test criteria.

Using this generator to test several Lua source code files shows that the developed unit test generator can help the unit testing process. It was expected that the unit test generator can improve productivity, quality, consistency, and abstraction of unit testing process.

Keywords—**unit test, code generator, Lua**

I. INTRODUCTION

Every software needs to meet its specified requirements. A software system need to be tested in order to determine whether it has met its requirement or not. Software testing process is an important step in the software development lifecycle [1], but often it cost a big portion of the project's budget and efforts. Test code productions is a time consuming process as it has to be done for all of the source codes and needs to be updated once a change was made to a source code. The budget and time limit may leads to poorly done testing process or worse, the testing process is not conducted. This problem can be solved by using code generator [2] to speed up the software testing process.

Lua is a language that was developed at PUC-Rio University's academic laboratory as a tool for in-house software development but somehow was adopted by several industrial projects around the world and is now widely used in the game industry [3, 4]. Lua is a fast, lightweight, embeddable scripting language. Although Lua is primarily a procedural language, it can be, and frequently is, used in several different programming paradigms, such as functional, object-oriented, goal-oriented, concurrent programming, and also for data description [5]. It has been used in many industrial applications with focuses on embedded systems and games. There are available some tools to help the testing of Lua source code such as LuaUnit and busted unit testing framework, but unlike other popular scripting language like JavaScript, Python, and Ruby, Lua does not have any unit test generator developed to help its software testing process.

Based on those problem mentioned above, on this research we propose an automatic code generation to speed up the process and helps avoiding repetition. The generated code is a unit testing code for Lua programming language. The code will comply with the standard of code structure for unit testing using LuaUnit or busted framework.

II. FOUNDATION AND RELATED WORKS

A. Software Testing

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended [6]. Search-based software testing is the use of metaheuristic optimizing search technique, such as a Genetic Algorithm, to automate or partially automate a testing task; for example the automatic generation of test data [7, 8].

Automatic test generation tools produce some valuable results that could facilitate the creation of a test suite. Automatic tools are really fast, high score in code coverage and mutation score, and it could help the developer creating unexpected scenarios and adding a further abstraction level to tests creation [9].

B. Related Works

There are several researches conducted on test automation. Some research are focused on automatic test data generation using search-based technique using source code as the main input value. Hollander [10] created JTestCraft, an automatic test generator for java language. It use candidate sequence

978-1-4673-8430-8/15/$31.00 ©2015 IEEE

search to search for method invocation sequence that can change the behavior of the method under test. Alshraideh [11] propose a tool for automatic JavaScript program testing using genetic algorithm and code instrumentation for branch coverage and mutation analysis. Vivanti, Mis, Gorla, and Fraser [12] develop a unit test generator with coverage criteria based on data-flow. Mairhofer, Feldt, and Torkar [13] created RuTeG, a tool for automatic test Ruby source code. It combines an evolutionary search for test cases that give structural code coverage with a learning component to restrict the space of possible types of inputs. While some other research make another approach by providing predefined test data set [14] and using MID (model-implementation description) specification, high-level petri nets, to generate test code [15].

III. Design and Implementation

In this section we introduce the code generator developed in this research, Lua Unit Test Generator (LUTG). Fig. 1 shows the basic structure of the code generator. LUTG use code munging as its active code generation model [2], where it take a file as the input, collect some information, and then generate the output code in another file.

LUTG consist of four main component *source code file analyzer*, *test data manager*, *test data generator*, and *unit test code generator*. Each of these component will be described below.

A. Source code file analyzer

The source code analyzer collect and extract some information from the source code that is required to generate unit test code. The source code file used as an input for this module must be a Lua module that contain one or more functions. The list of information that is collected by this module includes: (1) address to the source code file, (2) the original source code, (3) abstract syntax tree, (4) instrumented source code, (5) functions and its parameters list, (6) local module name, and (7) total branch of each function.

First, this module collect path of the source code file from user. Then, it read all of the file's content and parse it to generate an AST. The representation of the AST is a nested table complying the format specified by MetaLua [16]. Using this AST, we can extract the rest of the required information: instrumented source code, functions and its arguments list, local module name and total branch of each function.

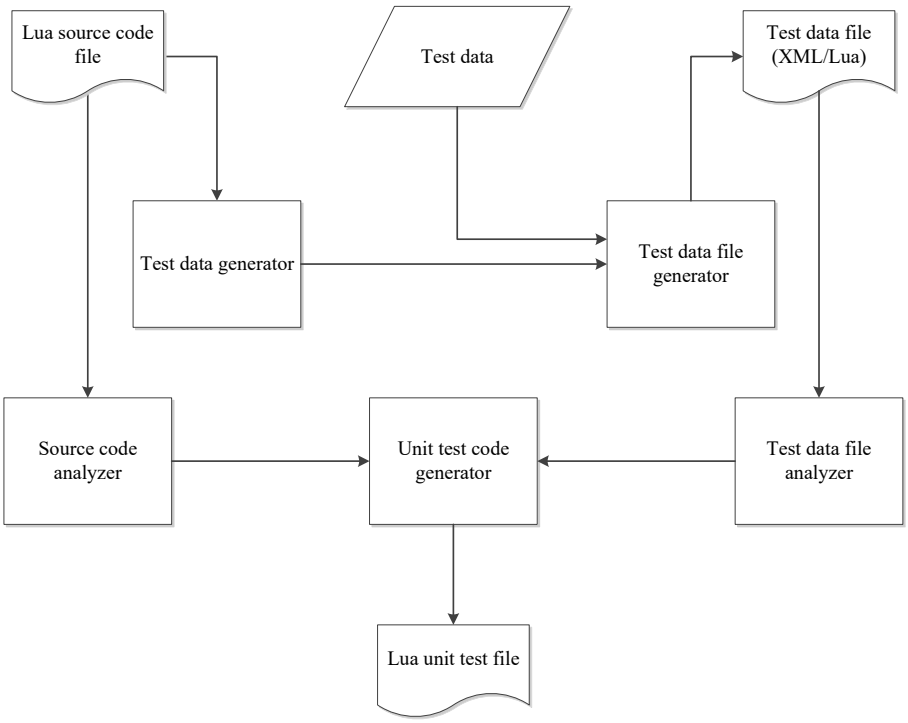

Fig. 1 Lua unit test generator scheme

B. Test data manager

This module manage test data information and handle the read and write process on test data file. Test data should contain the following information: (1) user selected unit test framework, (2) function under test's name, (3) test function group's name, (4) test function's name, (5) test setup (object initialization and other method invocation), (6) function under test parameters list, (7) input value for each parameter, expected return value, and (8) assertion type.

Other than test data, this module also manage mock data. A mock data consists of the following information: (1) user selected mock framework, (2) mock function's name, (3) mock function call input values, and (4) mock function call return value.

LUTG can help the end user to collect all of those information using wizards, test data and mock data have its own wizard. All of those information can be saved into a test data file in some file format. Currently there are two file format supported by this code generator: Lua and XML, other file format can be added by extending this module. Code 1 is an example of test data in Lua file format.

```
local source_testcase = {_framework = "luaunit"}
source_testcase["add"] = {
  testgroup_generated = {
    tc1= {args= {a=2, b=3}, exp= 5, assertType= "Equals"}
  }
}
return source_testcase
```

Code 1 Lua test data file

And Code 2 shows the same test data in XML format.

```
<test _framework = "luaunit">
<function name="add">
  <testCaseGroup name ="testgroup_generated">
    <testCase name="tc1">
      <args>
        <arg name="a">2</arg>
        <arg name="b">3</arg>
      </args>
      <exp>5</exp>
      <assertType>Equals</assertType>
    </testCase>
  </testCaseGroup>
</function>
</test>
```

Code 2 XML test data file

C. Test data generator

The test data generator module produces input value for every function inside the module under test. This module implement evolutionary search technique to generate appropriate test data. The selected evolutionary search technique is genetic algorithm. Genetic algorithm consist of several processes which is initialization, fitness evaluation, crossover, mutation, and reinsertion.

The genetic algorithm starts with a random initialization of population, which is then evolved using crossover and mutation to find fitter individual. The test data selection is based on the contribution of an individual to the test criteria, which is full branch coverage. Only individual that increase the number of branch coverage that will be included on the final set of generated test data.

Individual's chromosome contains information about input values for function under test. The values was decoded into binary string to ease the crossover and mutation processes. The chromosome contains a number of genes, one gene for one parameter on function under test. Each gene contains two information, the first 3 bit will determine the type of generated value and the next n bit contains the value that will be used as an input for specific parameter on function under test.

To get the information about branch coverage, the module under test are instrumented and executed using the generated test data. The result of the execution is then evaluated with fitness function.

$$F = B_{exec} / B_{tot} \quad (1)$$

Where B_{exec} is the number of branch covered in the current execution and B_{tot} is the number of total branch in the function under test.

The test data generator module currently can only generate Lua primitive data type values. Table 1 shows the default values range that can be generated by this generator. The value can be changed to get different input range. Another data type can also be added for better search result.

TABLE I GENERATED VALUES RANGE

Type	Bit	Range between	Description
Integer	16	-32767 and 32767	Two complements
Float	32	-3.402823×10^{38} and 3.402823×10^{38}	IEEE 754 single-precision binary floating-point format: binary32
String	7	ASCII code 32 and 126	7 bit / char
Table	32	-	Array of integer and string
Boolean	1	True or false	Boolean value

The selection process use roulette wheel method. Each individual's chance of being selected are proportionally to its fitness value, the bigger its fitness value the bigger chance it will be selected. The crossover process use uniform crossover. The chromosome of two individuals will be switched at several positions to increase the diversity of its offspring. The mutation process is rarely executed. Bit with value of 1 will be changed to 0, and vice-versa. Table 2 shows the default parameter used on the genetic algorithm to generate test data.

TABLE II GENETIC ALGORITHM'S PARAMETERS

Parameter	Value
Population size	40
Selection method	Roulette wheel
Mutation rate	0.002
Crossover rate	0.7
Crossover method	Uniform

The generator will keep on generating and collecting test data until full branch coverage is reached or if the branch coverage value is not increasing for a specific amount of generations.

D. Unit test code generator

The unit test code generator is the core of LUTG. It takes source code file and test data file as inputs to produce unit test file for the specific source code. The unit test code will comply the code structure of user's selected unit test framework and mock framework. Currently there are two unit testing framework that can be selected, LuaUnit and busted, and one mock framework which is mockagne. Another unit test and

mock framework can be added by extending unit test framework module and mock framework module respectively.

IV. Experiment

In this section, we want to test the applicability of LUTG on helping the testing process of Lua program. We conduct a small experiment to show how LUTG can make unit testing process for Lua program faster and easier. The experiment is using a module that contain two functions, the first function check if a string contain two or more consecutive upper case letter and the second function convert numerical grade to letter grade in term of academic grading system. Code 3 shows the modul for this experiment.

```
local demo = {}

function demo:twoConsUpper(a)
   local result
   if(string.find(a, "%u%u")) then result = false
   else result = true end
   return result
end

function demo:score(a)
   local result = "E"
   if a >= 0 and a <= 100 then
      if a >= 0 and a < 40 then result = "E"
      elseif a >= 40 and a < 55 then result = "D"
      elseif a >= 55 and a < 70 then result = "C"
      elseif a >= 70 and a < 85 then result = "B"
      else result = "A" end
   else
      result = "error"
   end
   return result
end

return demo
```

Code 3 Demo module

Using the wizard mentioned before, we manually add the following test data on Table 3. The code generator will produce test data file for the demo module. In this scenario we choose LuaUnit as the unit testing framework and Lua as the test data file format.

TABLE III DEMO MODULE TEST DATA

Input value	**Expected return**	**Assertion**
Function demo: twoConsUpper		
“TEst”	True	assertEquals
“TeSt”	False	assertEquals
Function demo:score		
Input value	Expected return	
90	“A”	assertEquals
50	“D”	assertEquals

Code 4 shows the generated Lua test data file containing data from Table 3.

```
local demo_testcase = {_framework = "luaunit"}

demo_testcase["demo:score"] = {
   test_score = {
      test_score_1 = {
         args = {a = 90},
         exp = A,
         assertType = "assertEquals"
      },
      test_score_2 = {
         args = {a = 50},
         exp = D,
         assertType = "assertEquals"
      }
   }
}
demo_testcase["demo:twoConsUpper"] = {
   test_twoConsUpper = {
      test_twoConsUpper_1 = {
         args = {a = "TEst"},
         exp = true,
          assertType = "assertEquals"
      },
      test_twoConsUpper_2 = {
         args = {a = "TeSt"},
         exp = false,
         assertType = "assertEquals"
      }
   }
}

return {demo_testcase}
```

Code 4 Demo module test data

Using the source code and the generated test data file, we can generate the unit test code. This process will be handled by the unit test code generator module. The content of the unit test file is shown on Code 5.

```
_MUT = require "demo"
require "luaunit"

test_demo_score = {}
function test_demo_score:test_score()
   assertEquals(_MUT:score(90),"A")
   assertEquals(_MUT:score(50),"D")
end

test_demo_twoConsUpper = {}
function test_demo_twoConsUpper:test_twoConsUpper()
   assertEquals(_MUT:twoConsUpper("TEst"),true)
   assertEquals(_MUT:twoConsUpper("TeSt"),false)
end

lu = LuaUnit.new()
lu:setOutputType("text")
os.exit( lu:runSuite() )
```

Code 5 Unit Test Code for Demo Module

Next, we try to use the test data generator to add the test data set. This process will be handled by the test data generator module. Below is the graph showing the branch coverage of the function under test on each generation.

As shown on Fig. 2 and Fig. 3, the test data generator module can achieve full branch coverage for each function under test. Please note that this process is not always give a satisfactory result because it depends on the complexity of the function. The range of possible input value may also affect the result.

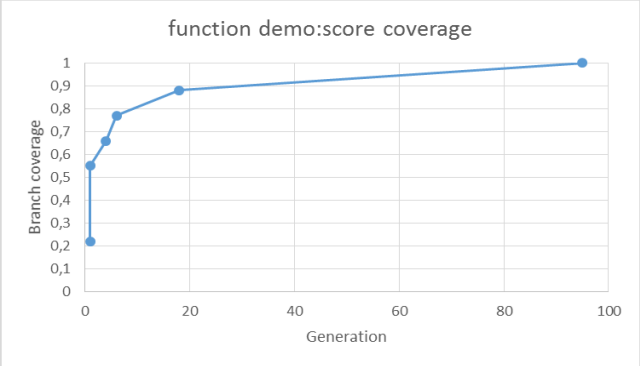

Fig. 2 Code coverage for demo:score function

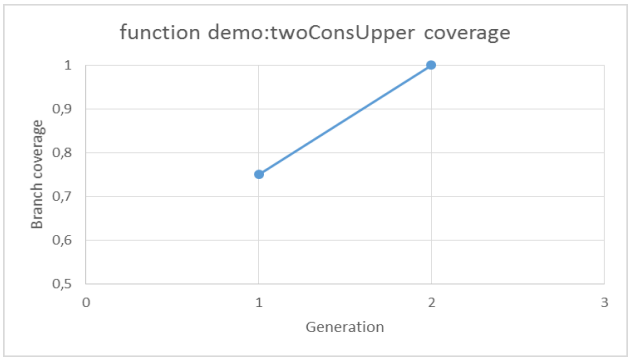

Fig. 3 Code coverage for demo:twoConsUpper

V. Discussion

In this research we propose LUTG, an automatic unit test code generator for Lua programming language. The code generator helps unit testing process of Lua programs by automating some of the processes such as generating test data using genetic algorithm, generating test data file, and generating unit test code.

Some weakness on this code generator lies on its test data generation process. The test data generator cannot generate test data for code written using object oriented paradigm yet. This feature is still in development. Besides that, the test data generation process is not always give satisfactory result on branch coverage. This can be caused by the complexity of the function under test or if the range of valid input domain is really narrow. The test data generator uses Lua source code as a reference in generating the test data. Therefore, error code can produce error test data.

The applicability of LUTG has been shown using a experiment. The experiment use a simple module to be tested with the help of LUTG. We realize that this small experiment may not reflect the real-world condition, but it can shows how LUTG can helps unit testing for code written in Lua language.

VI. Conclusion

The proposed tool, LUTG, has been proven to help the testing process of Lua program. It speed up and simplify the unit testing process. It also increases consistency of the unit test code compared to manually created unit test code, because it uses template to generate all unit test code.

The test data generator module helps tester to define test data, but the generated test data are based on the actual execution of the module under test. Even that it may provide a full branch coverage, it may not comply with the specification and may not reveal existing bug in the source code. The tester must check if the generated code has comply with the specification or not. Combination of automatically generated test data and manually inputted test data are the key to create the best test data set.

Acknowledgment

The researcher would like to express their gratitude to Roberto Ierusalimschy for the great Lua language and many code snippet implementation in Lua, Paul Kulchenko for helping the integration of LUTG on ZeroBrane Studio Lua IDE, and the anonymous referees for their valuable and helpful comments and suggestions in improving this research.

References

[1] Naik, K., & Tripathy, P. (2011). *Software testing and quality assurance: theory and practice*. John Wiley & Sons.

[2] Herrington, J. (2003). *Code generation in action*. Manning Publications Co. Greenwich, USA

[3] Ierusalimschy, R. (2013). *Programming in Lua*, 2nd edition. PUC-Rio, Brazil.

[4] Ierusalimschy, R., De Figueiredo, L. H., & Celes Filho, W. (2005). The Implementation of Lua 5.0. *J. UCS*, *11*(7), 1159-1176.

[5] Ierusalimschy, R. (2010). Programming with multiple paradigms in lua. In *Functional and Constraint Logic Programming* (pp. 1-12). Springer Berlin Heidelberg.

[6] Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

[7] McMinn, P. (2004). Search-based software test data generation: a survey. *Software testing, Verification and reliability*, *14*(2), 105-156.

[8] McMinn, P. (2011, March). Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on* (pp. 153-163). IEEE.

[9] Bacchelli, A., Ciancarini, P., & Rossi, D. (2008, October). On the effectiveness of manual and automatic unit test generation. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on* (pp. 252-257). IEEE.

[10] Hollander, D., M. A. (2010). *Automatic unit test generation* (Doctoral dissertation, TU Delft, Delft University of Technology).

[11] Alshraideh, M. (2008). A complete automation of unit testing for javascript programs. *Journal of Computer Science*, *4*(12), 1012.

[12] Vivanti, M., Mis, A., Gorla, A., & Fraser, G. (2013, November). Search-based data-flow test generation. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on* (pp. 370-379). IEEE.

[13] Mairhofer, S., Feldt, R., & Torkar, R. (2011, July). Search-based software testing and test data generation for a dynamic programming language. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation* (pp. 1859-1866). ACM.

[14] Adi, T. N. (2012). *Unit Test Code Generator for JavaScript Based on QUnit Framework*. (Master Thesis, School of Electrical Engineering and Informatics. Bandung Institute of Technology).

[15] Xu, D. (2011). A tool for automated test code generation from high-level Petri nets. In *Applications and Theory of Petri Nets* (pp. 308-317). Springer Berlin Heidelberg.

[16] Fleutot, F. Metalua-Static Meta-Programming for Lua.