Abstract-In software industry, the quality of software  
products should became a primary concern. One of many  
elements that affects to the software quality is testing.  
Creating a lot of test cases consumes a lot of time and  
tedious work. These issues can be solved by create a  
generator that automates the generation of input test data.  
Go language, or Golang for short, is a new programming  
language that developed by Google and has been used by  
many companies for their projects. However, Golang  
currently does not have any test generator that is able to  
generate test data with code coverage adequacy.  
In this research, we propose a process generation of test  
cases using Genetic Algorithm for Go language. We also  
have developed a tool based on this process generation.  
The tool does not only produce test data, but also the test  
code. By automating the generation, it is expected to  
reduce the amount of effort and time to generate manually.  
We have evaluated this process using eight different  
Golang source code. We found that the process and tool  
have facilitated the process of unit testing.

I. INTRODUCTION  
Software quality is one of the standards that the owners  
have to concern. Software testing is one of many way that  
affects the quality of software. In testing process,tester creates  
a lot of test cases for a unit under test, so unexpected results  
can be found [1]. But, 50% of total software development cost  
is used for testing process [2]. Creating a lot of test cases  
conswnes a lot of time and tedious work because this work is  
repetitive. This cost could be reduced by automate the  
generation of input data to the tested program [3].  
An adequacy of test cases can be determined by many  
techniques, such as mutating the tested code (mutation  
testing), calculation of branch coverage, measuring it by code  
coverage and so on. The code coverage shows how many lines  
or block are executed for a given test cases while the tests are  
running automatically.

Go language, also called Golang for short, has a testing  
tool called go test. This tool helps developers to testing  
units. The go test tool has a mode called "cover" to establish  
code coverage for a given test case set. While using this mode,  
go test tool creates an instrwnented code for code under test,  
and then compiles it and runs the test [4].  
In this research,we have proposed a method for generating  
a set of test cases that fulfilling code coverage by utilizing the  
go test tool with its "cover" mode.

III. DESIGN AND IMPLEMENTAnON  
In this section, we explain the architecture of the tool that  
we have developed, called Goceng (stands for Go Test Data  
and Code Generator). Goceng has four main components:  
analyzer,test data generator,test code generator,and executor.  
Fig. 1 shows the basic structure of the tool.  
The explanation as follows (Fig. 1):  
1) The tool receives source code of the program being tested.  
The source code is then analyzed by the analyzer to be  
collected the information within the source code, like  
declaration of functions, imports, types (structs), and  
constants (or enums).  
2) Test data generator component generates some input  
values based on the information (like what value types  
that must be generated for the input parameters of the  
function being tested) that was collected by the analyzer.  
3) Test code generator component generates test code file.  
The file contains a declaration of array of test case, an  
invocation of the function under test with inputs from  
each test case in the array using an iteration, and an  
examination of every return values that obtained from the  
invocation of the function under test.

4) Since the tool need to perform evolutionary generation in  
order to achieve full code coverage, the executor runs the  
generated test code using go test tool to collect coverage  
profile for the given set of test data. If the coverage result  
is not satisfied, the generation of the test data process is  
recurred.

The following is the detail of each process. First, Goceng  
receives path of source code with some generation  
configuration and manual test cases (written in JSON syntax).  
The tool then loads source codes from a given path and parses  
them into AST. With AST representation,we can extract some  
needed information,like defmed structs, functions (input and  
output parameter types), constants (especially enums), and  
their locations (file path).

Because the initial test data is generated randomly, the  
generated test cases may not pass some conditions, like  
x==490 (for int), y==3.5 (for float), or str=="2300 days"  
(for string), so we may not achieve a full code coverage. To  
handle these conditions, the analyzer collects these basic  
literal values from all conditional statements (like if-else,  
switch -case and for statement). These literal values are then  
appended to the randomly generated input values in test data  
generation process.  
For string type, we need to collect basic literal values  
with more advanced way. String comparison usually use  
equality (using '==' symbol), or use included string's  
function in strings package, like HasPrefix(),  
HasSuffix(), Contains(), and EqualFold(). Basic literal  
values are also collected from these invoked functions in the  
code.

Enums in Golang is declared like a constants declaration,  
but using an extended type (defined by type Mylnt int). The  
analyzer also finds and collects these declarations, so the test  
data generator produces the name of constants (enum) instead  
of literal values.  
After some data get collected by the analyzer, then test  
data generator generates input values for tested unit (function).  
Since we want the generated test cases to achieve a code  
coverage, test generator performs a genetic algorithm for test  
data generation.  
Test generator is divided into two, test data generator and  
test code generator. Test data generator performs test case  
generation using GA based on code coverage adequacy. Since  
we utilize go test with "cover" mode, we use different way  
to generate test cases using GA from approach introduced by  
Pargas et al. in [7]. It is very time consuming if we implement  
that approach because go test performs code instrumentation  
and then compiles it before the test run.  
We have tried to fix these issues by changing the  
implementation of genetic algorithm in our tool. We propose  
that one population may contain many individuals, where each  
individual contains many test cases. Gen in our approach is  
one test case for once invocation of the function under test.  
Each input values in test data are called locus. This locus will  
be crossover with other locus from different individual (see  
Fig. 2, Parent 1 shows individual, a row of XI,I, YI,I and ZI,1  
shows gen and Xl,l is locus). The difference is that each  
individual in our approach is a set of test cases, while  
approach that proposed in [7] each individual is a test case  
itself. With this technique is expected to reduce time  
consuming while collecting the code coverage reports.  
Test code generator produces testing code. The inline-code  
generation technique is implemented for the code generation,  
as explained in [14]. This generation technique helps us in  
development and maintenance of tool because it provide a  
template that may change dynamically every time it is  
executed, but the pattern is same. This component produces 3  
types of testing code by its role:  
1) Test code that is used for collect code coverage. It only  
contains generated test cases (only input values)  
declaration and function invocation for every given test  
case.  
2) Test code that is used for collect result values from the  
function's return statement.  
3) Final test code,where the test case contains not only input  
values but also output values.  
Population is initialized by performing a random value  
generation. After that, each individual (which is a set of test  
cases) is evaluated its fitness by run the test using go test  
command. If there is no individual with expected fitness,  
reproduction is performed. Reproduction is done by  
performing crossover to every test case of selected 2 parents  
(this process is shown in Fig. 2), per input parameters, where  
parents are selected using roulette wheel and random  
technique. The reproduction process produces 2 offsprings.  
Each of the offsprings are then mutated with a small  
probability, by generating new mutator (each input value is

generated randomly), and then perform crossover like  
crossover process with 2 parent before. If offsprings contain a  
literal value collected by the analyzer in analyze process, the  
value must not be changed. Offsprings are then evaluated their  
fitness. If any offspring of two is stronger than the weakest  
individual in population, then the weakest individual in  
population should be changed with this offspring. This process  
repeats until all path is pass by any individual (a set of test  
cases) or execution time is over (defined by user).

IV. EXPERIMENT  
In this section, we have tested the applicability of Goceng  
in Golang unit testing process, by evaluating the suitability of  
generated value types with input parameters like primitive  
types, compound types (struct types) and enums by user and  
generated test case that achieves full code coverage. We have  
tested the tool on BMI function,shown in Fig. 3.  
We have performed 10 times generation at maximum  
crossover duration time is 5 minutes (300,000 ms), shown in  
Table 1. One of ten tests cannot reach full code coverage.  
Three of ten tests performed genetic algorithm process to  
reach full code coverage, and six of ten tests reached full  
branch coverage by generating value randomly (at population  
initialization process). The same process were then repeated  
for another 7 source code with different complexity.

V. DISCUSSION  
We have proposed Goceng as automatic test data and code  
generator for Golang. The tool helps unit testing process of  
Golang programs by automating some of the processes, such  
as generating test data that uses code coverage adequacy using  
genetic algorithm, the generated test data is saved to JSON  
file, generating test code by generated test data or saved  
generated test data from JSON file.  
Some weaknesses of Goceng lies on its test data  
generation process. Since we utilize go test tool while  
instrumenting code and collecting code coverage results, it is  
hard to determine which path would be passed by generated  
test case.  
Full code coverage can be reached easily when branch  
conditions are simple, means there are no reassign variable  
statement or arithmetic operation for each input parameters  
before or on conditional statement. We have tried to fix this  
issue by attaching the feature in our tool that lets the users  
create their own test cases in JSON format. The tool then read  
it,translate it and append it to the test code.

VI. CONCLUSION  
Our proposed tool can help the testing process of Golang  
program. It simplify the unit testing process by automate test  
data generation with search-based testing approach and  
generate test code by template.  
Furthermore, Goceng can generate not only primitive but  
also compound type (struct type) defined by user. Goceng  
checks all return values from function - Golang function can  
return more than one value - and generate enum by write a  
constant name instead of literal value in the test code - enum in  
Golang is created and defined like a constants declaration and  
defmition.  
At this stage, Goceng could not generate test data for the  
code that was written in object-oriented. Currently we are still  
developing research on how to generate test data for unit in  
object-oriented paradigm. In the future, we plan to add this  
feature in Goceng. We are hoping that this research would  
lead to another testing tool for Go language.