Pembangkit *Test Case* Untuk Pengujian Perangkat Lunak Menggunakan Metode *Basis Path*

Bayusandya Tresnayatna[1], Ir. Sri Widowati, M.T.[2], Iman Lukmanul Hakim, M.M.[3]

[1,2,3]Fakultas Informatika, Universitas Telkom, Bandung
[4]Divisi Digital Service PT Telekomunikasi Indonesia
[1]bayusandya@students.telkomuniversity.ac.id, [2]sriwidowati@telkomuniversity.ac.id, [3]imanlhakim@gmail.com

Abstrak

Basis path merupakan salah satu metode pengujian perangkat lunak unit testing. Dengan metode ini kita dapat menghitung jumlah dari setiap percabangan pada alur logika.[3] Dengan menghitung jumlah jalur percabangan maka dapat pula menentukan test case untuk digunakan dalam sekenario pengujian unit testing. Melakukan pekerjaan tersebut tidak mudah, belum lagi ketika menemukan algoritma yang kompleks seperti terdapatnya fungsi nested, sehingga dirasa perlu dibuatkan alat yang dapat membangkitkan test case secara otomatis dengan mengimplementasikan metode basis path, alat ini dibangun menggunakan teknik parsing agar mengetahui seluruh komponen percabangan pada source code kemudian dibuatkan algortima penghitungan rekursif pada method untuk dapat menghitung dan mendefinisikan setiap kondisi percabangan. Test case yang dihasilkan oleh tool pembangkit kemudian dilakukan pengujian terhadap beberapa studi kasus dengan bermacam kondisi dan dilakukan pengujian false test. Kesimpulan yang didapat adalah membangun alat yang dapat membangkitkan test case secara otomatis dengan mengimplementasikan metode basis path dapat memudahkan pekerjaan untuk melakukan pengujian perangkat lunak unit testing.

Kata Kunci : Software Testing, Unit Testing, Basis Path, Test Data, Test Case

Abstract

Base path is one of the unit testing software testing methods. With this method we can calculate the number of each branch in the logic flow.[3] By calculating the number of branch lines, we can also determine the test case to be used in the unit testing for test scenario. Doing the work is not easy when finding complex algorithms such as the nested functions, so that it is necessary to make a tool that can generate test cases automatically by implementing base path methods, this tool is built using parsing techniques to find all branching components then a recursive calculation algorithm is made for the method to be able to calculate and define each branching condition. Generated test case tested against several case studies with various conditions and tested against false test. The conclusion is build a tool that can generate test cases automatically by implementing the base path method that can facilitate the work to do unit testing in software testing.

Keywords: Software Testing, Unit Testing, Base Path, Test Data, Test Case

1. Pendahuluan

Latar Belakang

Banyak *developer* yang ingin mengetahui apa *bug/error* yang terdapat dalam sistem yang mereka bangun, cara untuk menemukan *bug/error* adalah dengan melakukan *software testing*. *Unit Testing* adalah metode pengetesan yang sangat handal dalam pengujian perangkat lunak pada suatu unit terkecil program. [4] Pengujian ini menggunakan *path coverage* untuk memeriksa dan mengevaluasi setiap alur logika pada sistem pemrograman. Hal tersebut tentunya sulit untuk dikerjakan jika sisitem yang sedang dalam masa pengetesan memiliki tingkat kerumitan percabangan yang tinggi.[3] *Basis path* adalah metoda yang dapat diimplementasikan dalam pekerjaan ini, konsep dasar *basis path* ialah hanya berfokus menghitung pada statement percabangan saja, sehingga alur pemrograman yang bersifat *sequence* idak perlu di telusuri lagi (*skipped*). *Basis path* dapat menghitung jumlah *test case* sekaligus dengan nilainya dengan menganalisis *source code* yang di translasikan kedalam bentuk *control flow graph* (CFG) dan menghasilkan *cyclomatic complexity* (CC) untuk setiap jalur percabangan pada *graph*, kemudian akan didapatkan *independent path* (IP) yang nantinya menjadi penentu jumlah *test case* dan macam *test* data-nya.[3][5] Namun karena sulitnya untuk menentukan *test case* dari suatu *source code* atau studi kasus, maka pendefinisian *test case* harus dilakukan oleh *tool* secara otomatis.[1][3] Topik ini menarik untuk di angkat sebagai bahan penelitian untuk membangun *tool* yang dapat menghasilkan *test case*, sehingga kesulitan dalam menentukan *test data* teratasi.

Dalam membangun *tool*, penulis menggunakan teknik *parsing* untuk dapat melakukan pengecekan ke setiap *token* pada *source code* komponen program. Seperti layaknya manusia ketika mencari kemudian menemukan dan mencatat setiap *statement* percabangan. *Source code* akan di konversi ke bentuk *tree graph* agar mesin dapat mengenali struktur *code*, kemudian melakukan analisis dengan algoritma yang dapat menghitung setiap iterasi jika ditemukan percabangan. Maka lengkaplah syarat *tool* untuk dapat membangkitkan *test data*, kemudian menguji hasilnyadan dilakukan analisis sebagai tahap akhir dari pengerjaan penelitian ini.

Topik dan Batasannya

Unit Testing sangat perlu dilakukan demi menjamin kualitas sebuah perangkat lunak, namun kasus muncul ketika proses tes terdapat kesulitan dalam menentukan nilai *test data* untuk seknario test, maka diperlukannya pendefinisian *test case* dengan metode *basis path* untuk mempermudah proses melakukan pengujian perangkat lunak. Sehingga dapat dirangkum pada rumusan masalah sebagai berikut:

1. Bagaimana mengimplementasikan basis path testing untuk men-generate test case.
2. Membangun tool yang dapat diimplementasikan dalam pengujian unit testing.

Tujuan

Tujuan dari penelitian ini adalah membangun *tool* yang dapat diimplementasikan dalam pengujian perangkat lunak, dengan membangkitkan *test case* untuk digunakan dalam membuat sekenario pengtesan.

Organisasi Tulisan

Pada bagian selanjutnya, bagian 2 menjelaskan mengenai studi literature perihal metode dan teknik yang digunakan, pada bagian 3 menjelaskan secara detil mengenai sistem yang di bangun, pada bagian 4 membahas evaluasi dari sistem dan bagian 5 merupakan kesimpulan dari penelitian yang telah dilakukan.

2. Studi Terkait

2.1 Unit Teting

Unit Testing merupakan metode pengujian perangkat lunak, pada dasarnya pengujian dilakukan pada bagian unit dari suatu sistem, termasuk juga pengujian *white box*, artinya melihat *source code* untuk melakukan evaluasi *input output* yang sesuai terhadap fungsi dan prosedur secara individual. Dalam pemrograman berorientasi objek biasa dilakukan pengujian hanya pada suatu kelas. *Unit testing* dilakukan pada masa pembangunan oleh *tester developer* untuk memastikan tidak adanya kesalahan fungsi saat sebelum fungsi tersebut di panggil.[5] Pengujian selalu dilakukan berdasarkan *test case* (kasus uji) agar pengujian ter-struktur dan tidak redundan. *Test case* dapat dibuat jika tersedianya data tes.

2.2 Test Case

Test Case, adalah data berbentuk suatu kondisi atau parameter pada percabangan seperti ”*if*, *while*, dan *for*”, berisi informasi yang dapat menentukan arah jalur program berjalan. biasanya format penulisan *Test Case* dapat berupa seperti contoh berikut:

```
if (n < 90) {
    eksekusi = 'A';
    } else {
      eksekusi = 'B';
```

Gambar 1. Kondisi pada “if”

(n < 90) menandakan bahwa ”n” harus memiliki nilai kurang dari 90 untuk mengeksekusi A, kemudian pada kondisi lainnya nilai ”n” harus lebih besar dari 90 untuk mengeksekusi B, data ini lah yang sering di sebut dengan *test data*.

2.3 Test Data

Test data atau data test adalah kumpulan informasi yang didapatkan dari *source code*, berisi inputan data agar fungsi dapat melakukan eksekusi dan menentukan arah *path*. *Test data* diperoleh dengan melihat setiap statement kondisi pada percabangan. Contoh pada *java code* sebagai berikut pada metod `IfElseDemo`:

```
class IfElseDemo {
    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else {
            grade = 'B';
      }
        System.out.println("Grade = " + grade);
    }
}
```

Gambar 2. If Else Demo

Dapat diketahui bahwa terdapat kondisi dimana jika (`testcore >= 90`) maka fungsi akan mengeksekusi kondisi A, jika selain itu maka masuk ke kondisi B, informasi ini merupakan *test data* yang dibutuhkan untuk membuat sekenario test. Hasilnya dilakukan pencatatan pada setiap inputan dan kondisi. Contoh dapat dilihat dalam format tabel begai berikut:

Tabel 1. Test Data

ID	Test Data	Catatan
1.a	input: 76	masuk kondisi B
1.b	input: 95	masuk kondisi A

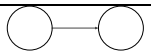
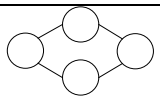
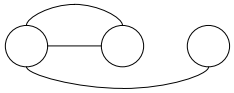
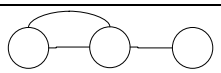
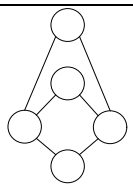
2.4 Basis Path

Pengujian metode basis path meganalisa kedalam program domain untuk memperoleh test data yang sesuai, berbasis Cyclomatic Complexity (CC) yang membantu tester untuk mempekirakan jumlah kasus uji yang diperlukan. (CC) dapat diperoleh dengan membuat Control Flow Graph (CFG) merupakan gambaran umum dari program domain namun dalam bentuk graph, dengan menghitung jumlah nodes dan edges dengan persamaan sebagai berikut: V(G) = e – n + 2, dimana e adalah notasi untuk setiap sisi dan n merupakan notasi node. Nilai V(G) merupakan jumlah minimum test data yang diperlukan. Dengan metode lain cara memperoleh nilai (CC) dapat dilakukan dengan menghitung jumlah kondisi seperti (if, else, while, dll). [2]

2.5 Control Flow Graph

Control flow graph terdiri dari node dan edge, ada beberapa notasi yang menggambarkan bentuk struktur dari model sistem sebagai berikut :

Tabel 2. Tipe – tipe notasi

Tipe	Notasi
Sequence	
IF	
While	
Until	
Case	

Node dapat menggambarkan beberapa statement tergantung dengan kondisi, baik itu secara sekuensial, perulangan dan lain-lain, dapat berupa percabangan. Notasi sisi (edge) menggambarkan aliran kontrol.[4]

2.6 Cyclomatic Complexity

Dari CFG, cyclomatic complexity model program dapat dibuat dengan menggunakan rumus berikut :

$$V(G) = e - \mathrm{n} + 2$$

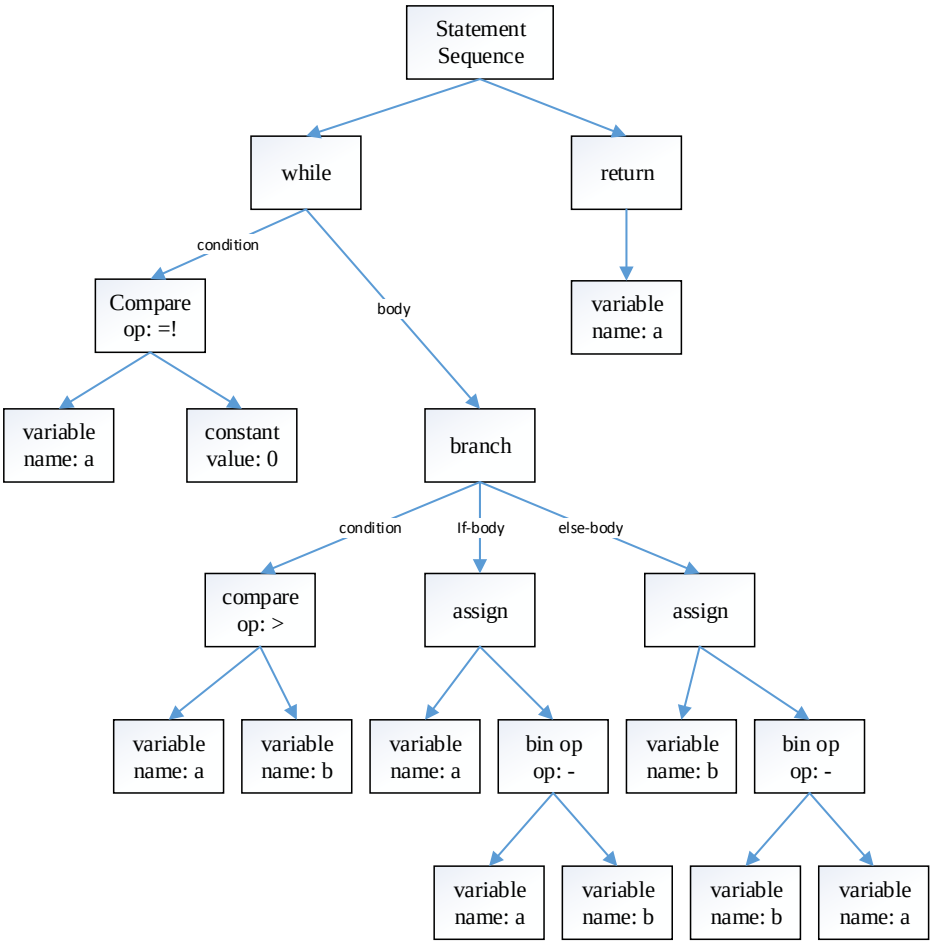
Penjelasan dari rumusnya adalah V(G) : cyclomatic complexity, E : total jumlah edge, dan N : total jumlah node.[3]

2.7 Independent Path

Hasil perhitungan cyclomatic complexity merupakan jumlah independent path, menunjukkan jumlah minimal pengujian yang harus dijalankan. IP merupakan setiap jalur program yang menunjukkan satu set pemerosesan kondisi baru.[4] Independent path pada CFG harus melewati satu edge yang belum pernah dilewati oleh path sebelumnya yang dimulai dari awal node hingga ke akhir.[3]

2.8 Antlr4

ANTLR (*ANother Tool for Language Recognition)* adalah generator parser yang dibuat untuk membaca, memproses, mengeksekusi, atau menerjemahkan teks terstruktur atau file biner. Fungsi dari *Antlr4* adalah sebagai parser dalam proses mengevaluasi *code*.



Gambar 3: Contoh AST dari algoritma Euclid

Abstract Syntax Tree (AST) adalah model *graph antlr4* yang di *generates* dari *source code*. Jika dilihat struktur *graph* nya terlihat sangat kompleks untuk sebuah algoritma *euclid*, itu disebabkan karena AST dapat melakukan parsing hingga ke tingkat *atom* sehingga meski *rule basenya* terlihat ringkas, namun sbetulnya itu adalah algoritma yang kompleks.

Mesin akan membaca mulai dari atas kemudian ke kiri dan kebawah terlebih dahulu struktur ini biasa disebut dengan "*tree walker*". Terdapat dua jenis metode "*walking tree"*, yang pertama dapat menggunakan *visitor* dan *listener*, keduanya memiliki fungsi yang sama, yakni dapat mengenali setiap tokenisasi yang terdapat dalam *parse tree*, namun keduanya memiliki aturan yang berbeda tergantung pada kebutuhan *grammar* serta rule dan *output* yang diinginkan.

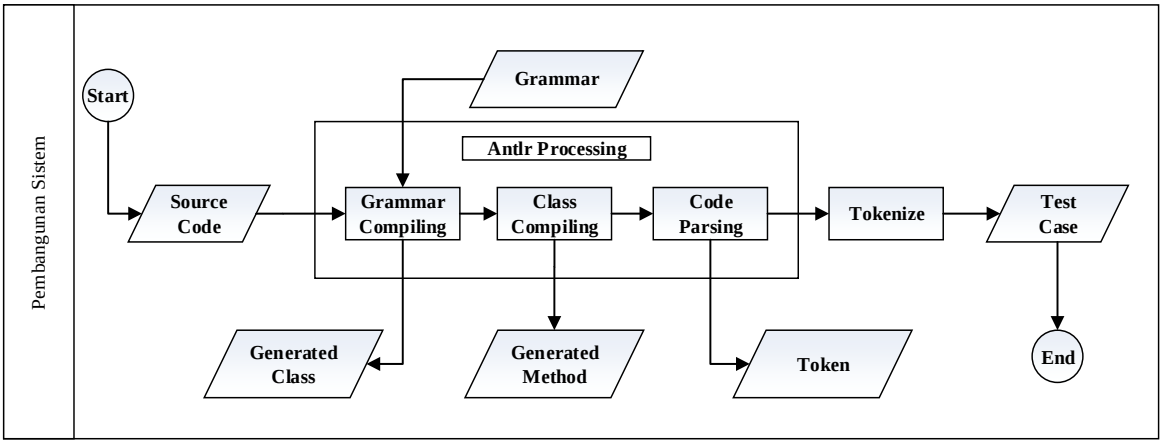
Model inilah yang dapat dibaca oleh mesin parsing untuk berjalan dan me-*capture* setiap statement yang dibutuhkan. Antlr4 membutuhkan *grammar* bahasa pemrograman untuk setiap bahasa pemrogramannya kemampuan inilah yang memungkinkan kita dapat membuat bahasa pemrograman sendiri dengan rule dan grammar yang dapat pula ditentukan sendiri. *Output* dari *antlr* sangat beragam, dapat dirangkum sebagai berikut:

1. Compile grammar output: Lexer, Parser, Listener, Base listener, visitor, Base visitor.
2. Compile grammar + rule: Parse Tree Graph, Token.

Dengan metode yang dihasilkan, dapat digunakan sebagai pemanggilan fungsi pada *method* yang kita buat, kemudian dapat pula ditentukan, *output* apa yang ingin kita lihat.

3. Sistem yang Dibangun

Pada bagian ini dijelaskan proses serta metode saat melakukan penelitian. Berikut adalah gambaran keseluruhan dari pembangunan sistem:



Gambar 3. Diagram Alur Pembangunan Sistem

Pada Gambar 3 diatas menunjukan diagram alur pembangunan sistem dalam membangkitkan *test case* yang dimulai dengan memasukan data *source code,* kemudian dilakukan pengujian dan analisis pada bab 4. Berikut adalah penjelasan dari masing – masing proses:

3.1 Antlr Processing

Antlr adalah *tool* yang dapat melakukan *parsing*, proses *setup*-nya memerlukan tahapan yang panjang dan saling beruhubungan, jika salah satu tahapan terdapat kesalahan maka hasil dari proses *parsing* akan salah. Proses dimulai dari pembuatan *grammar*, *parsing* terhadap *grammar*, melakukan *compiling* terhadap *generated java class* dan *parsing* terhadap *source code*.

3.2 Grammar Compiling

Grammar Compiling merupakan tahap parsing *grammar* yang telah dibuat, fungsi dari *grammar* adalah kamus besar suatu bahasa pemrograman agar mesin mengerti dan mengenali karakter dari bahasa yang dijadikan studi kasus. Tahap ini menggunakan *java programming compiler*, *compile* dapat dilakukan dengan cara membuat perintah pada *console* dengan *syntax*: *java –cp [antlr] [grammar] -package [package]*. Setelah perintah di eksekusi maka dihasilkan beberapa kelas dengan nama depan persis seperti nama *grammar*, kelas yang akan di bangkitkan dari sebuah *grammar* antara lain adalah: "*JavaLexer.tokens","* *JavaParser.java", "JavaBaseLitener.java", "JavaLexer.java", "JavaListener.java & Java.token".* Didalam kelas - kelas tersebut terdapat method yang harus dilakukan *compile* sekali lagi, proses tersebut akan dijelaskan pada bab 3.3.

3.3 Class Compiling

File kelas *java* yang telah di bangkitkan kemudian dilakukan compiling dengan perintah *javac*, contoh *syntax* kali ini aadalah dengan parameter sebagai berikut: *javac [antlr] [JavaGrammar*.java]*. Dengan perintah tersebut *antlr* akan membangkitkan seluruh *java file.*

3.4 Code Parsing

Pada tahap ini adalah proses *antlr* melakukan *parsing* pada *source code* dengan *syntax* yang dijalankan melalui *console*, hal yang dibutuhkan untuk melakukan parsing pada tahap ini adalah: *source code*, *grammar*, dan *antlr.jar*, contoh untuk melakukan parsing pada *code* adalah dengan memberikan perintah: *java –cp antlr4 org.antlr4 [grammar] [source code]*. Penelitian ini menggunakan studi kasus pada bahasa pemrograman *java*, maka *grammar* dan *source code* disesuaikan. Perintah yang dilakukan tersebut adalah untuk membangkitkan seluruh data *token* untuk tahapan selanjutnya yang akan di jelaskan pada bab 3.5.

3.5 Tokenize

Tokenisasi adalah tahap pembuatan algoritma penghitungan jumlah statement *if*, *while* atau *for*, serta penghitungan *independent path* berdasarkan *source code* yang telah diparsing sebelumnya sudah dijelaskan pada bab 3.4. Pada dasarnya *antlr* merupakan *java library* yang hanya melakukan *parsing* dan mengumpulkan data yang disimpan aman dalam *method* yang terbagi dalam kelas – kelas *java*, algoritma di desain untuk dapat mengolah data hasil dari *parsing* kemudian melakukan rekursif untuk menghitung jumlah *statement* dan menampilkan *output* program. Pada bab 3.6 akan dijelaskan seluruh keluaran yang dihasilkan oleh pembangkit.

3.6 Output Program

Hasil keluaran dari pembangkit adalah: *test case*. Sebelum dilakukan pengujian, *test case* terlebih dahulu dirubah kedalam format tabel seperti contoh pada **lampiran 2**, agar dapat lebih mudah untuk mendefinisikan *test data* maka dirubah lagi kedalam format tabel seperti contoh pada **lampiran 4**, pada tahap ini sekenario tes sudah di bentuk maka siap untuk di eksekusi terhadap *source code* sebagai bahan penugjian dan evaluasi yang akan dijelaskan pada bab 4.

4. Evaluasi

Bagian ini menjelaskan dua sub-bagian, hasil pengujian dengan tiga studi kasus, serta analisis dari hasil pengujian.

4.1 Hasil Pengujian

Dari tiga studi kasus pengujian hasilnya adalah selalu sesuai dengan nilai dari *expected result*, masing masing dapat dilihat pada **lampiran 5, 11, dan 16**. Untuk menguji apakah sistem yang dibangun sudah benar, sengaja dilakukan *false test* dengan memasukan *test data* yang tidak sesuai dengan hasil pembangkit terhadap studi kasus, hasilnya sesuai seperti apa yang diekspektasikan, pada studi kasus hasilnya mengembalikan keluaran *error*, contoh hasil pengujian tersebut dapat dilihat pada **lampiran 6**.

4.2 Analisis Hasil pengujian

Pengujian dilakukan dengan memasukan *test data* yang di dapat dari *tool* pembangkit, hasilnya program pada s*ource code* dapat mengeksekusi fungsinya dengan baik dan mengeluarkan *output*-an yang sesuai dengan apa yang diharapkan sebelumnya (*expected results*). Untuk memastikan kebenaran dari hasil program kemudian dilakukan pengujian *false test* hasilnya program pada *source code* tidak dapat melakukan eksekusi dengan baik (terjadi *error*) seuai dengan (*expected result*):

Tabel 3. Pengujian false test menggunakan JUnit

SEKENARIO FALSE TEST			
Test ID	**Deskripsi**	**Ekspected Result**	**Actual Result**
FT01	Pengecekan "if" pada kondisi true maka masukan huruf: “false test”.	Program berhenti dengan output: “error”.	Program output: “error”.
FT02	Pengecekan "while" pada kondisi loop maka masukan string: “false test”.	Program mengeksekusi dengan output: “error”.	Program output: “error”.
FT03	Pengecekan "For" pada kondisi loop maka masukan angka: “5”. & “A, C, F, G, H”.	Program mengeksekusi dengan output: “error”.	Program output: “error”.

Pengujian *test case* dilakukan dengan *tool JUnit* yang dapat membantu dalam mengeksekusi fungsi.

5. Kesimpulan

Kesimpulan dari penelitian ini adalah sebagai berikut:

1. Setiap algoritma pada sistem pasti memiliki percabangan, baik itu bentuk “*while, if,* atau *for* dll”. Maka untuk melakukan pengujian hanya perlu fokus pada setiap percabangan, karena besar kemungkinan terjadinya *defect*, inilah sebabnya metode *basis path* diimplementasikan pada pengujian perangkat lunak *unit box testing*.

2. Mendefinisikan percabangan pada suatu *source code* cukup sulit. Dengan *tool automated test case generator* ini dapat diketahui jumlah dan nilai data uji untuk dijadikan sekenario dalam melakukan *unit testing*.

Saran yang diberikan untuk penelitian selanjutnya adalah sebagai berikut:

1. Algoritma yang akan dilakukan pengujian sebaiknya sudah dalam bentuk *code progam* yang sudah baik secara fungsi dan dapat dilakukan *run* program. Contoh: pada kelas java sudah tersedia *method main*. hal ini akan memudahkan penelitian pada tahap pengujian, karena pengujian dilakukan dengan cara otomasi oleh JUnit.

2. Jika ingin melakukan parsing terhadap suatu algoritma (bahasa pemrograman sendiri) disarankan untuk membuat *grammar* yang standar, jika *grammar* tidak proper maka akan terjadi kendala ketika mendesain algoritma agar dapat melakukan rekursif.

Daftar Pustaka

[1] Aldeida Aleti, Lars Grunske, 2014, Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm, Malaysia, Monash University, University of Stuttgart.

[2] Yeresime Suresh, Santanu Ku Rayh., 2013, A Genetic Algorithm based Appoarch for Test Data Generationaing Path Testing, India, Department of Computer Science and Engineering, National Institute of Technology.

[3] K.Vivekanandan, T.Megala, P.Chandini, 2016, Automatic Generation of Basis Test Path Using Clonal Selection Algorithm, India, Computer Science and Engineering Pondicherry Engineering College Pondicherry.

[4] Zhang Guangmei, Chen Rui, Li Xiaowei, Han Congying, 2005, The Automatic Generation of Basis Set of Path for Path Testing, Cina, Shan Dong University of Science and Technology.

[5] Chi-Kuang Chang, Nai-Wei Lin, 2015, UTGen: a Black-Box Method-Level Unit-Test Generator for JUnit Test-Platform, Taiwan, Department of Computer Science and Information Engineering, National Chung Cheng University.

[6] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, Hong Mei, 2009, Prioritizing JUnit Test Cases in Absence of Coverage Information, Cina, School of Electronics Engineering and Computer Science, Peking University.

[7] Pablo Lamela Seijas, Simon Thompson, Miguel Ángel Francisco, 2016, Model extraction and test generation from JUnit test suites, Inggris, University of Kent.

[8] Mohammad Wahid, Dr. Abdullah Almalaise, 2011, JUnit Frameworl: An Interactive Approach for Basic Unit Testing Learning in Software Engineering, Malaysia, Faculty of Science and Information Systems Universisti Teknologi, Jeddah Saudi Arabia, Faculty of Computing and Information Technology King Abdulaziz University.

[9] Marie-Claude Gaudel, 2017, Formal Methods for Software Testing, Perancis, LRI, Univ. Paris-Sud, CNRS, CentraleSup´elec Universit´e Paris-Saclay.

[10] Antônio Mauricio Pitangueiraa, Rita Suzana P. Maciela, Márcio Barros, 2014, Software requirements selection and prioritization using SBSEapproaches: A systematic review and mapping of the literature, Brazil, BarrosbaFederal University of Bahia, Computer Science Department.