Jurnal Ilmu Komputer Agri-Informatika

Tersedia secara *online* di:
http://journal.ipb.ac.id/index.php/jika

Volume 5 Nomor 1 halaman 40 – 50
ISSN: 2089-6026

Instrumentasi Kode Program Secara Otomatis untuk *Path Testing*

Automatic Source Code Instrumentation for Path Testing

RADEN ASRI RAMADHINA FITRIANI[1*], IRMAN HERMADI[1]

Abstrak

Pengujian perangkat lunak yang kompleks secara keseluruhan akan memakan waktu yang lama dan membutuhkan sumber daya manusia yang banyak. Mengotomasi bagian dari pengujian akan membuat proses ini menjadi lebih cepat dan mengurangi kerawanan akan kesalahan. Pada penelitian ini, telah dibangun sebuah aplikasi untuk membangkitkan kemungkinan jalur-jalur dari sebuah program yang dapat dijadikan dasar untuk membangkitkan data uji agar data uji yang digunakan untuk pengujian dapat mewakili semua kemungkinan. Selain itu, aplikasi ini juga dapat melakukan penyisipan *tag-tag* sebagai instrumentasi ke dalam kode program secara otomatis untuk memonitor jalur mana yang dilalui ketika diberikan masukan data uji. Hasil penelitian menunjukkan bahwa eksekusi menggunakan aplikasi menghabiskan waktu rata-rata 0.228 detik. Sedangkan jika dilakukan secara manual menghabiskan waktu rata-rata 383.28 detik atau 6 menit 23 detik.

Kata Kunci: *Control Flow Graph*, *Cyclomatic Complexity*, Instrumentasi, *Matlab, Path Testing*

Abstract

Complex software testing as a whole will take a long time and require a lot of human resources. Automating parts of testing will make the process faster and reduce the vulnerability errors. In this study, an application has been developed to generate possible path from a program that can be used as a basis to generate test data in order to test data used for testing can represent all possibilities. In addition, this application can also insert tags as instrumentation into the program code automatically to monitor which path is passed when given input test data. The results showed that the executable using the application spent time average 0228 seconds. Whereas if done manually spent time average 383.28 seconds or 6 minutes 23 seconds.

Keywords: Control Flow Graph, Cyclomatic Complexity, Instrumentation, Matlab, Path Testing

PENDAHULUAN

Pengujian adalah serangkaian proses yang dirancang untuk memastikan sebuah perangkat lunak melakukan apa yang seharusnya dilakukan. Proses ini bertujuan untuk menemukan kesalahan pada perangkat lunak (Myers *et al.* 2012). *Path testing* merupakan salah satu metode pengujian struktural yang menggunakan kode program untuk menemukan semua jalur yang mungkin dapat dilalui program dan dapat digunakan untuk merancang data uji. Metode ini memastikan semua kemungkinan jalur dijalankan setidaknya satu kali (Basu 2015). Pada proses pengujian ini diperlukan penanda yang dapat memberikan informasi cabang mana yang dilalui untuk memonitor jalur mana yang diambil oleh sebuah masukan pada saat eksekusi program. Proses menyisipkan tanda tersebut disebut instrumentasi. Biasanya tanda tersebut disisipkan tepat sebelum atau sesudah sebuah percabangan (Tikir dan Hollingsworth 2011).

Idealnya, pengujian dilakukan untuk semua kemungkinan dari perangkat lunak. Namun, untuk menguji perangkat lunak yang kompleks secara keseluruhan akan memakan waktu yang lama dan membutuhkan sumber daya manusia yang banyak. Kumar dan Mishra (2016) mengatakan bahwa pengujian perangkat lunak menggunakan hampir 60% dari total biaya pengembangan perangkat lunak.

Hermadi (2015) melakukan penelitian membangkitkan data uji untuk *path testing* menggunakan algoritme genetika. Dalam penelitian tersebut, Hermadi membangkitkan *Control*

[1]Departemen Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam, Institut Pertanian Bogor, Bogor 16680
[*]Penulis Korespondensi: Surel: *radenasrirf@gmail.com*

Flow Graph (CFG) dan semua kemungkinan jalur sebagai *target path* sehingga data uji yang akan dibangkitkan harus mewakili semua *target path.* Setelah jalur terbantuk, Hermadi melakukan instrumentasi pada kode program dengan menyisipkan kode program untuk memonitor jalur mana yang dilalui dari suatu data input ketika *test data generator* dijalankan. Kode program yang digunakan sebagai studi kasus dalam penelitian tersebut adalah kode program yang dibangun dengan menggunakan bahasa *Matlab*. Proses pembangkitan CFG, pembangkitan semua kemungkinan jalur, dan instrumentasi masih dilakukan secara manual sehingga membutuhkan banyak waktu dan rawan akan kesalahan ketika program sudah semakin besar.

Pada penelitian ini, akan dibangun sebuah perangkat lunak untuk membangkitkan semua kemungkinan jalur dari sebuah program dan melakukan penyisipan *tag-tag* sebagai instrumentasi ke dalam kode program secara otomatis.

METODE

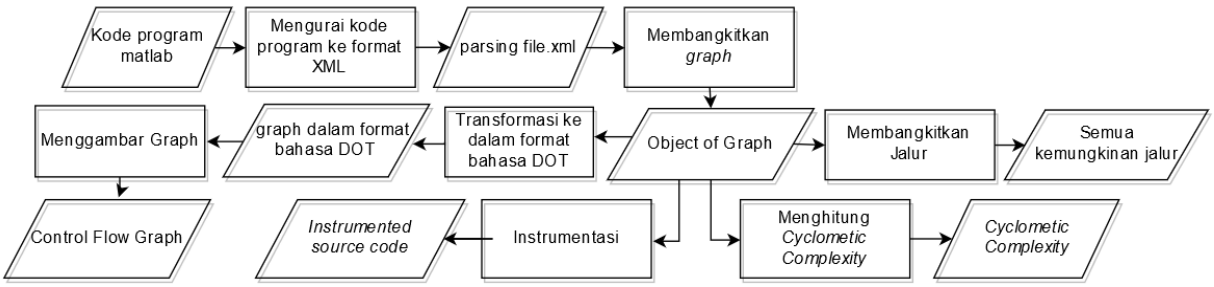
Penelitian yang dilakukan terbagi menjadi 4 tahapan, yaitu analisis, perancangan, implementasi, dan pengujian.

Analisis Kebutuhan

Pada tahap ini dimulai dari membaca literatur terkait dan mendefinisikan kebutuhan dari aplikasi yang akan dibangun. Selain itu, pada tahapan ini juga dilakukan pengumpulan beberapa contoh program yang akan digunakan dalam penelitian. Contoh program yang akan digunakan pada penelitian ini diperoleh dari penelitian yang dilakukan oleh Hermadi (2015).

Perancangan Sistem

Pada tahap ini ditentukan bagaimana perangkat lunak akan dibangun. Ilustrasi arsitektur sistem dapat dilihat pada Gambar 1.



Gambar 1 Arsitektur sistem

1. Kode Program

Kode program *Matlab* akan dibaca sebagai inputan. Seperti bahasa pemrograman lainnya, *Matlab* memiliki beberapa kontrol struktur. Kontrol struktur adalah perintah dalam bahasa pemrograman yang digunakan dalam pengambilan keputusan. *Matlab* memiliki empat kontrol struktur, yaitu *IF-ELSE-END*, *SWITCH-CASE*, *FOR*, dan *WHILE* (Houcque 2005).

2. Mengurai Kode Progam ke Format XML

Penguraian kode program Matlab menjadi file dengan format XML dilakukan dengan menggunakan *library Matlab-PARSER* yang dibangun oleh Suffos (2015).

3. Membangkitkan *Graph*

Salah satu cara untuk membaca dan menulis dokumen XML pada framework .NET dan C# yaitu dengan menggunakan kelas *XMLDocument* yang terdapat dalam *namespace System.XML*.

4. Membangkitkan Jalur

Jalur dibentuk dengan cara menelusuri objek graph yang sudah dibentuk sebelumnya. Jika edge memiliki tipe true atau false, maka jalur yang dibangkitkan akan ditambahkan informasi

cabang yang dilalui. Jalur akan melalui (T) ketika melalui edge yang memiliki tipe true dan (F) ketika melalui edge yang memiliki tipe *false*. Jalur yang dibentuk ketika melalui perintah pengulangan seperti *FOR* dan *WHILE* akan dibatasi maksimal satu kali pengulangan. Hal ini dilakukan karena *path testing* bertujuan memastikan setiap *node* dilalui setidaknya satu kali sehingga jalur yang melaui pengulangan *node* lebih dari satu kali tidak dibutuhkan.

5. Transformasi ke Dalam Format Bahasa *DOT*

Graph yang sudah terbentuk akan ditransformasikan ke dalam bentuk format bahasa permrograman *DOT*. Bahasa *DOT* adalah bahasa yang digunakan untuk mengambar *graph* berarah (Ganser *et al* 2015).

6. Memvisualisasikan *Graph* dalam bentuk CFG

Control Flow Graph (CFG) adalah *graph* berarah yang merepresentasikan aliran dari sebuah program. Setiap CFG terdiri dari *nodes* dan *edges*. *Nodes* merepresentasikan perintah. Sedangkan *edges* merepresentasikan kontrol transfer antar *nodes* (Watson dan McCabe 1996). Setelah *file* dengan format bahasa *DOT* terbentuk, CFG akan divisualisasikan dengan menggunakan *library Graphviz*. *Graphviz* merupakan perangkat lunak *open source* untuk visualisasi grafik berarah (Ellson *et al* 2003).

7. Menghitung *Cyclometic Complexity*

Cyclomatic complexity merupakan suatu sistem pengukuran yang ditemukan oleh Watson dan McCabe (1996) untuk menentukan banyaknya *independent path* dan menunjukan tingkat kompleksitas dari suatu program. *Independent path* adalah jalur yang melintas dalam program yang sekurang-kurangnya terdapat kondisi baru. Perhitungan *Cyclomatic Complexity* dapat dilihat pada persamaan (1) berikut:

$$V(G) = E - N + 2 \quad (1)$$

dengan E menunjukkan jumlah *edges* dan N menunjukkan jumlah *nodes*.

8. Instrumentasi

Instrumentasi dilakukan dengan cara menambahkan dulu variabel keluaran bernama *traversedPath*. Variabel ini digunakan untuk menyimpan informasi *node* mana saja yang dilalui ketika diberikan inputan dengan nilai tertentu. Kemudian, setiap sebelum dan sesudah *node* percabangan, dilakukan penyisipan kode program berupa perintah untuk memasukkan nilai *node* yang dilalui. Sehingga ketika program tersebut dijalankan, akan menghasilkan keluaran tambahan bernama *traversedPath.*

Implementasi

Tahapan ini adalah melakukan implementasi dari tahap sebelumnya ke dalam bentuk aplikasi web. Aplikasi ini akan dibangun dengan menggunakan bahasa pemrograman C# dan menggunakan *IDE Microsoft Visual Studio Ultimate* 2013. CFG akan divisualisasikan dengan menggunakan *library Graphviz.Net. Graphviz.Net* adalah pembungkus C# untuk generator grafik *Graphviz* yang dibangun oleh Dixon (2013).

Pengujian

Tahapan ini adalah melakukan evaluasi dari tahapan implementasi. Evaluasi dibagi menjadi dua bagian, yaitu uji efisiensi dan uji validasi. Uji efisiensi dilakukan dengan membandingkan waktu eksekusi yang dilaukan secara manual dengan waktu eksekusi oleh aplikasi. Pengujian manual akan dilakukan dengan meminta dua orang yang sudah memiliki pengalaman dalam pemrograman sebagai sampel untuk melihat berapa lama waktu yang dibutuhkan untuk melakukan hal yang sama dengan aplikasi. Waktu yang diperoleh dari pengujian melalui aplikasi adalah waktu eksekusi yang dijalankan di dalam lokal komputer tanpa menggunakan koneksi internet.

Uji validasi dilakukan dengan cara membandingkan hasil yang ada pada penelitian sebelumnya dengan hasil yang dikeluarkan oleh aplikasi. Pada penelitian sebelumnya yang dilakukan oleh Hermadi (2015), *graph* yang dibangun adalah *graph* yang hanya

menggambarkan notasi percabangan dan tanpa penomoran *node*. Agar dapat dibandingkan dengan hasil yang dikeluarkan oleh aplikasi, *graph* yang ada pada penelitian sebelumnya direpresentasikan ke dalam bentuk *adjacency list* terlebih dahulu secara manual.

HASIL DAN PEMBAHASAN

Analisis Kebutuhan

Sebelumnya sudah terdapat beberapa program yang dapat membangkitkan CFG seperti *Eclipse Control Flow Graph Generator*, tetapi *library* tersebut hanya dapat digunakan di eclipse dan hanya membangkitkan CFG dari kode program java (Alimucaj 2009). Aplikasi yang akan dibuat dalam penelitian ini adalah aplikasi yang dapat membangkitkan CFG, membangkitkan semua kemungkinan jalur, menghitung *cyclomatic complexity*, dan melakukan instrumentasi. Terdapat 10 contoh program yang akan digunakan pada penelitian ini. Contoh program yang akan digunakan dapat dilihat pada Tabel 1.

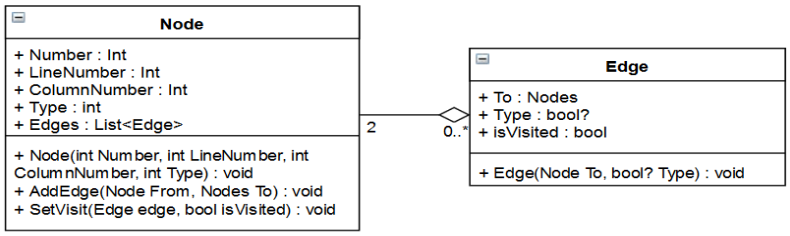
Tabel 1 Contoh program uji

No	Program Uji	Nama	Deskripsi
1	Triangle Ahmed	tA2008	Mencari tipe dari segitiga.
2	Minimaxi Ahmed	mmA2008	Mencari nilai minimal dan maksimal dari bilangan dalam array
3	Quotient Bueno	qB2002	Menghitung hasil dan sisa hasil bagi dari dua buah bilangan bulat
4	Binary Ahmed	binA2008	Mencari indeks sebuah bilangan dalam array
5	Fitness Minimaxi Hermadi	fmH2015	Menghitung fungsi fitness dari fungsi minimaxiAhmed2008
6	Bubble Ahmed	bubA2008	Mengurutkan bilangan menggunakan metode bubble sort
7	Insertion Ahmed	iA2008	Mengurutkan bilangan menggunakan metode insertion sort
8	Expint Bueno	eB2002	Fungsi eksponensial
9	Flex Gong	fG2011	Sebuah utilitas unix yang diambil dari situs GNU
10	Gcd Ahmed	gA2008	Menghitung GCD atau pembagi dua bilangan terbesar

Perancangan Sistem

1. Perancangan *Class diagram*

Class diagram dibangun untuk menggambarkan struktur sistem dari segi pendefinisian class dan hubungan antar class. Perancangan class diagram dapat dilihat pada Gambar 2**.**



Gambar 2 Perancangan *Class Diagram*

Dalam sebuah *class node* terdapat informasi nomor *node*, nomor baris dan nomor kolom dari kode program yang akan digunakan untuk melakukan instrumentasi. Selain itu, terdapat tipe dari perintah tersebut apakah termasuk percabangan, pengulangan, perintah biasa, atau akhir dari sebuah perintah. Selain itu, terdapat list *edge* yang berisi *node* tujuan dan tipe dari *edge* yang digunakan jika terdapat percabangan *true*, *false*, atau hanya garis penghubung biasa. Setiap *edge* memiliki atribut *isVisited* yang digunakan untuk menandakan apakah garis penghubung tersebut sudah dilalui atau belum.

Implementasi

Aplikasi dibangun dengan menggunakan bahasa pemrograman C# dan menggunakan *IDE Microsoft Visual Studio Ultimate 2013*.

1. Kode Program

Kode program tA2008 dapat dilihat pada Gambar 3. Program ini digunakan untuk mencari jenis dari segitiga jika diketahui panjang dari setiap sisinya. Pada kode tA2008 terdapat perintah *IF-THEN-ELSE* bersarang sebanyak tiga tingkat.

```
1   function type = triangle(sideLengths)
2           A = sideLengths(1); % First side
3           B = sideLengths(2); % Second side
4           C = sideLengths(3); % Third side
5           if ((A+B > C) && (B+C > A) && (C+A > B))
6                   if ((A ~= B) && (B ~= C) && (C ~= A))
7                           type = 'Scalene';
8                   else
9           if (((A == B) && (B ~= C)) || ((B == C) && (C ~= A)) || ((C == A) && (A ~= B)))
10                                  type = 'Isosceles';
11                          else
12                                  type = 'Equilateral';
13                          end
14                  end
15          else
16                  type = 'Not a triangle';
17          end
18  end
```

Gambar 3 Kode Program tA2008

2. Mengurai Kode Progam ke Format XML

Ketika ditemukan perintah *IF* maka akan dibentuk sebuah elemen *<if></if>*. Lalu untuk bagian memenuhi kondisi *IF* akan disimpan di dalam elemen *<If.IfPart></If.IfPart>*. Ekspresi dari kondisi *IF* akan disimpan di dalam elemen *<IfPart.Expression> </IfPart.Expression>*. Perintah yang akan dilakukan ketika memenuhi kondisi *IF* akan disimpan dalam elemen *<IfPart.Statements></IfPart. Statements>*.

3. Membangkitkan *Graph*

Representasi objek dari kelas *graph* yang terbentuk dari kode program tA2008 dapat dilihat pada Gambar 4. *Graph* disimpan ke dalam struktur data *adjacency list* dari objek *node* yang dihubungkan oleh objek *edge*. Terbentuk *graph* yang terdiri atas 9 buah *nodes* dan 11 buah *edges* yang menghubungkan antar *nodes* tersebut.

4. Membangkitkan Jalur

Berikut merupakan semua kemungkinan jalur yang akan yang dapat dijadikan sebagai dasar dalam pembangkitan data uji. Kemungkinan jalur direpresentasikan dalam bentuk urutan nomor *node* dan terdapat informasi tambahan (T) ketika melalui *edge* yang memiliki tipe *true* dan (F) ketika melalui *edge* yang memiliki tipe *false.*

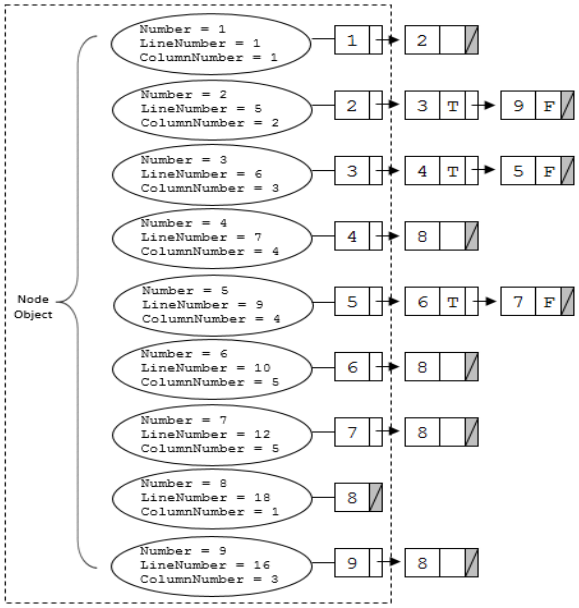
a. 1 2 (T) 3 (T) 4 8
b. 1 2 (T) 3 (F) 5 (T) 6 8
c. 1 2 (T) 3 (F) 5 (F) 7 8
d. 1 2 (F) 9 8

5. Transformasi ke Dalam Format Bahasa *DOT*

Transformasi ke dalam format bahasa *DOT* dilakukan dengan cara menelusuri objek *graph* yang sudah dibangun sebelumnya. Elemen yang didefinisikan dalam bahasa *DOT* adalah *edge* yang terdapat pada *graph* yang dibangun. Seperti yang dapat dilihat pada Gambar 5, jumlah baris sebanyak jumlah *edge* pada objek *graph* yang telah didefinisikan sebelumnya.

6. Memvisualisasikan *Graph*

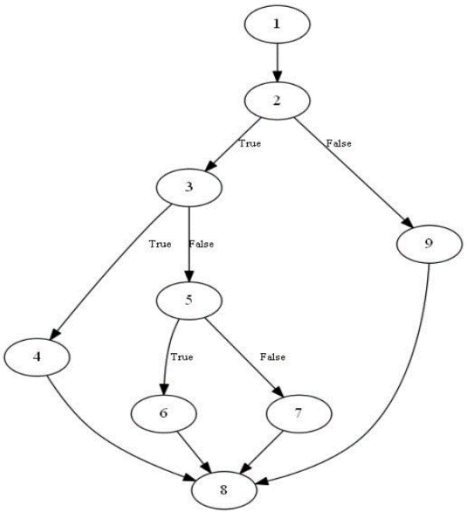
Setelah file dengan format bahasa *DOT* terbentuk, CFG divisualisasikan dengan menggunakan *library Graphviz.Net* dengan menginputkan bahasa *DOT* yang sudah dibentuk sebelumnya. Hasil visualisasi kode program tA2008 ke dalam CFG dapat dilihat pada Gambar 6.



Gambar 4 Representasi tA2008 dalam bentuk *Object Graph*

```
1   digraph                G                {
2   graph      [label=""       nodesep=0.8]
3   1->2;
4   2->3  [  label="True"    fontsize=10  ];
5   3->4  [  label="True"    fontsize=10  ];
6   3->5  [  label="False"   fontsize=10  ];
7   5->6  [  label="True"    fontsize=10  ];
8   5->7  [  label="False"   fontsize=10  ];
9   7->8;
10  6->8;
11  4->8;
12  2->9  [  label="False"   fontsize=10  ];
13  9->8;
14  }
```

Gambar 5 Representasi tA2008 dalam bahasa *DOT*



Gambar 6 CFG tA2008

7. Menghitung Cyclometic Complexity

Berdasarkan *graph* yang telah terbentuk dari kode program tA2008, dapat dilihat pada Gambar 4 bahwa jumlah *node* yang terbentuk adalah 9 dan jumlah *edge* yang terbentuk adalah 11. Sehingga hasil perhitungan *cyclomatic complexity* dapat dilihat pada persamaan dibawah ini.

$$V(G) = E - N + 2 = 4$$

Hasil perhitungan *cyclomatic complexity* dari kode program tA2008 adalah 4 yang menunjukan jumlah jalur dasar yang akan terbentuk.

8. Instrumentasi

Hasil kode program yang telah diinstrumentasi dapat dilihat pada Gambar 7. Kode program tA2008 awalnya hanya mengembalikan keluaran satu variabel bernama *type* yaitu menunjukan jenis dari segitiga ketika diberikan panjang dari ketiga sisi segitiga. Setelah dilakukan instrumentasi, kode program tA2008 akan mengembalikan keluaran dengan variabel tambahan bernama *traversedPath.* Sehingga ketika program tersebut dijalankan dengan inputan tertentu akan menghasilkan keluaran nilai *traversedPath* dan *type* seperti yang dapat dilihat pada Gambar 8.

```
function [traversedPath,type] = triangle(sideLengths)
        traversedPath = [];
        traversedPath = [traversedPath '1 ' ];
        A = sideLengths(1); % First side
        B = sideLengths(2); % Second side
        C = sideLengths(3); % Third side
        % instrument Branch # 1
        traversedPath = [traversedPath '2 ' ];
        if ((A+B > C) && (B+C > A) && (C+A > B))
                traversedPath = [traversedPath '(T) ' ];
                % instrument Branch # 2
                traversedPath = [traversedPath '3 ' ];
                if ((A ~= B) && (B ~= C) && (C ~= A))
                        traversedPath = [traversedPath '(T) ' ];
                        traversedPath = [traversedPath '4 ' ];
                        type = 'Scalene';
                else
                        traversedPath = [traversedPath '(F) ' ];
                        % instrument Branch # 3
                        traversedPath = [traversedPath '5 ' ];
                          if (((A == B) && (B ~= C)) || ((B == C) && (C ~= A))      || ((C   ==
A) && (A ~= B)))
                                traversedPath = [traversedPath '(T) ' ];
                                traversedPath = [traversedPath '6 ' ];
                                type = 'Isosceles';
                        else
                                traversedPath = [traversedPath '(F) ' ];
                                traversedPath = [traversedPath '7 ' ];
                                type = 'Equilateral';
                        end
                end
        else
                traversedPath = [traversedPath '(F) ' ];
                traversedPath = [traversedPath '9 ' ];
                type = 'Not a triangle';
        end
        traversedPath = [traversedPath '8 ' ];
end
```

Gambar 7 Hasil instrumentasi tA2008

```
>> [traversedPath, type] = triangle([3,4,4])

traversedPath =

    '1 2 (T) 3 (F) 5 (T) 6 8 '


type =

    'Isosceles'
```

Gambar 8 Hasil ekseskusi kode program tA2008 yang sudah diinstrumentasi

Tabel 2 Perbandingan waktu eksekusi secara manual dan menggunakan aplikasi

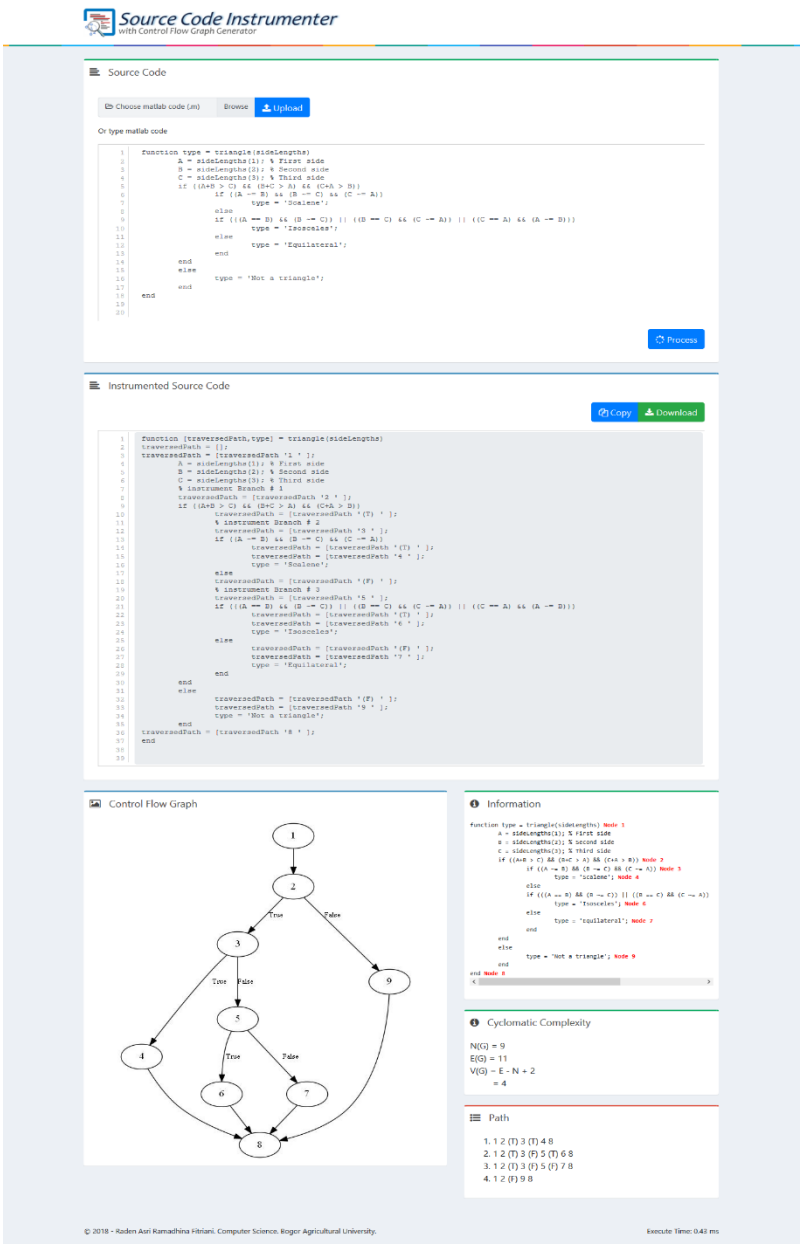
No	Nama Program	Waktu Eksekusi Aplikasi (detik)	Waktu Ekseskusi Manual (detik)		
			Penguji 1	Penguji 2	Rata-Rata
1	tA2008	0.19	-	-	-
2	mmA2008	0.26	558.23	407.90	483.07
3	iA2008	0.11	234.55	358.51	296.53
4	binA2008	0.48	384.64	526.59	455.62
5	bubA2008	0.11	448.64	207.97	328.31
6	eB2002	0.22	686.59	632.27	659.43
7	gA2008	0.21	492.93	293.01	392.97
8	qB2002	0.16	222.25	280.86	251.56
9	fmH2015	0.20	201.87	223.07	212.47
10	fG2011	0.34	335.92	403.27	369.60
Rata-Rata		0.22	396.18	370.38	383.28

Implementasi

Halaman implementasi antarmuka hasil dari proses yang telah dilakukan oleh aplikasi yang dapat dilihat pada Gambar 9.

Pengujian

Hasil perbandingan waktu yang dibutuhkan untuk melakukan pembangkitan secara manual dan oleh aplikasi dapat dilihat pada Tabel 2. Waktu yang dibutuhkan aplikasi untuk rata-rata adalah 0.228 detik. Sedangkan jika dilakukan secara manual akan menghabiskan waktu rata-rata 383.28 detik atau 6 menit 23 detik.



Gambar 9 Tampilan implementasi antarmuka hasil dari proses yang telah dilakukan oleh aplikasi

Tabel 3 menunjukkan perbandingan *adjacency list* yang dibangun berdasarkan pada penelitian sebelumnya dan *adjacency list* yang dibangun menggunakan aplikasi. Dari 10 program uji, bentuk *graph* yang terbentuk jika divisualisasikan dalam bentuk CFG sama. Perbedaan hanya terdapat pada label penomoran beberapa *node*.

Tabel 3 Perbandingan adjacency list manual dan menggunakan aplikasi

No	Nama Program	Adjacency List Manual	Adjacency List Aplikasi
1	tA2008	- 1 -> 2 - 2 -> 3 -> 5 - 3 -> 4 -> 6 - 5 -> 9 - 4 -> 7 -> 8 - 6 -> 9 - 7 -> 9 - 9 - 8 -> 9	- 1 -> 2 - 2 -> 3 -> 9 - 3 -> 4 -> 5 - 4 -> 8 - 5 -> 6 -> 7 - 6 -> 8 - 7 -> 8 - 8 - 9 -> 8
2	mmA2008	- 1 -> 2 - 2 -> 3 -> 8 - 3 -> 4 -> 5 - 4 -> 5 - 5 -> 6 -> 7 - 6 -> 7 - 7 -> 2 - 8	- 1 -> 2 - 2 -> 3 -> 8 - 3 -> 4 -> 5 - 4 -> 5 - 5 -> 6 -> 7 - 6 -> 7 - 7 -> 2 - 8
3	qB2002	- 1 -> 2 - 2 -> 3 -> 10 - 3 -> 4 -> 10 - 4 -> 5 -> 6 - 5 -> 4 - 6 -> 7 -> 10 - 7 -> 8 -> 9 - 8 -> 9 - 9 -> 6 - 10	- 1 -> 2 - 2 -> 3 -> 10 - 3 -> 4 -> 10 - 4 -> 5 -> 6 - 5 -> 4 - 6 -> 7 -> 10 - 7 -> 8 -> 9 - 8 -> 9 - 9 -> 6 - 10
4	binA2008	- 1 -> 2 - 2 -> 3 -> 9 - 3 -> 4 -> 5 - 4 -> 5 - 5 -> 6 -> 7 - 6 -> 8 - 7 -> 8 - 8 -> 2 - 9 -> 10 -> 11 - 10 -> 12 - 11 -> 12 - 12	- 1 -> 2 - 2 -> 3 -> 9 - 3 -> 4 -> 5 - 4 -> 5 - 5 -> 6 -> 7 - 6 -> 8 - 7 -> 8 - 8 -> 2 - 9 -> 10 -> 11 - 10 -> 12 - 11 -> 12 - 12
5	fmH2014	- 1 -> 2 - 2 -> 3 -> 4 -> 5 - 3 -> 6 - 4 -> 6 - 5 -> 6 - 6 -> 7 -> 10 - 7 -> 8 -> 9 - 8 -> 10 - 9 -> 10 - 10	- 1 -> 2 - 2 -> 3 -> 4 -> 5 - 3 -> 6 - 4 -> 6 - 5 -> 6 - 6 -> 7 -> 10 - 7 -> 8 -> 9 - 8 -> 10 - 9 -> 10 - 10
6	bubA2008	- 1 -> 2 - 2 -> 3 -> 8 - 3 -> 4 -> 7 - 4 -> 5 -> 6 - 5 -> 6 - 6 -> 3 - 7 -> 2 - 8	- 1 -> 2 - 2 -> 3 -> 8 - 3 -> 4 -> 7 - 4 -> 5 -> 6 - 5 -> 6 - 6 -> 3 - 7 -> 2 - 8
7	iA2008	- 1 -> 2 - 2 -> 3 -> 6 - 3 -> 4 -> 5 - 4 -> 3 - 5 -> 2 - 6	- 1 -> 2 - 2 -> 3 -> 6 - 3 -> 4 -> 5 - 4 -> 3 - 5 -> 2 - 6
8	eB2002	- 1 -> 2 - 2 -> 3 -> 4 -> 5 -> 6 -> 7 - 3 -> 11 - 4 -> 11 - 5 -> 11 - 6 -> 8 -> 11 - 8 -> 9 -> 10 - 9 -> 10 - 10 -> 6 - 11 - 7 -> 12 -> 13 - 12 -> 14 - 13 -> 14 - 14 -> 15 -> 20 - 15 -> 16 -> 18 - 16 -> 19 - 18 -> 19 -> 19 - 19 -> 18 - 19 -> 20 -> 21 - 20 -> 21 - 21 -> 14	- 1 -> 2 - 2 -> 3 -> 4 -> 5 -> 6 -> 11 - 3 -> 10 - 4 -> 10 - 5 -> 10 - 6 -> 7 -> 10 - 7 -> 8 -> 9 - 8 -> 9 - 9 -> 6 - 10 - 11 -> 12 -> 13 - 12 -> 14 - 13 -> 14 - 14 -> 15 -> 20 - 15 -> 16 -> 17 - 16 -> 19 - 17 -> 18 -> 19 - 18 -> 17 - 19 -> 20 -> 21 - 20 -> 21 - 21 -> 14
9	fG2011	- 1 -> 2 - 2 -> 3 -> 20 - 3 -> 4 -> 11 - 4 -> 5 -> 6 - 5 -> 7 - 6 -> 7 - 7 -> 8 -> 9 - 8 -> 10 - 9 -> 10 - 10 -> 12 -> 13 - 11 -> 10 - 12 -> 14 -> 15 - 14 -> 16 - 15 -> 16 - 16 -> 17 -> 18 - 13 -> 16 - 17 -> 19 - 18 -> 19 - 19 -> 2 - 20	- 1 -> 2 - 2 -> 3 -> 20 - 3 -> 4 -> 11 - 4 -> 5 -> 6 - 5 -> 7 - 6 -> 7 - 7 -> 8 -> 9 - 8 -> 10 - 9 -> 10 - 10 -> 12 -> 16 - 11 -> 10 - 12 -> 13 -> 14 - 13 -> 15 - 14 -> 15 - 15 -> 17 -> 18 - 16 -> 15 - 17 -> 19 - 18 -> 19 - 19 -> 2 - 20
10	gA2008	- 1 -> 2 - 2 -> 3 -> 4 - 3 -> 9 - 4 -> 5 -> 9 - 5 -> 6 -> 7 - 6 -> 8 - 7 -> 8 - 8 -> 4 - 9	- 1 -> 2 - 2 -> 3 -> 4 - 3 -> 9 - 4 -> 5 -> 9 - 5 -> 6 -> 7 - 6 -> 8 - 7 -> 8 - 8 -> 4 - 9

SIMPULAN

Penelitian ini berhasil membangun sebuah aplikasi yang dapat digunakan untuk melakukan instrumentasi secara otomatis, membangkitkan CFG, menghitung *cyclomatic complexity*, dan membangkitkan segala kemungkinan jalur yang dapat dilewati dari kode program *Matlab* untuk *path testing.*

Hasil penelitian menunjukkan bahwa kecepatan eksekusi jauh lebih cepat dibandingkan dilakukan secara manual sehingga dapat menghemat sumber daya dalam melakukan pengujian perangkat lunak.

SARAN

Penelitian selanjutnya diharapkan dapat mengakomodir bahasa lain selain Bahasa Matlab, terutama bahasa-bahasa yang banyak digunakan para pengembang perangkat lunak. Selain itu, dapat mempertimbangkan jalur logik kondisi majemuk dari sebuah percabangan.

Lebih jauh lagi, instrumentasi tidak hanya menyisipkan kode program untuk menyimpan informasi jalur yang dilalui, tetapi juga dapat dikustomisasi menggunakan *tag-tag* sesuai dengan kebutuhan. Seperti yang dilakukan oleh Hermadi (2015), selain menyimpan informasi jalur mana yang dilewati, instrumentasi yang dilakukan pada penelitian tersebut juga menyisipkan kode program untuk memanggil fungsi menghitung nilai *fitness* dari setiap data uji pada setiap posisi *node.*

UCAPAN TERIMA KASIH

Terima kasih penulis sampaikan kepada Bapak Prof Dr Ir Agus Buono, MSi MKom, Bapak Dr Wisnu Ananta Kusuma, ST MT, dan Ilham Tri Mulyawan, S.Komp yang telah memberikan saran dan masukan dalam penelitian ini serta kepada seluruh dosen dan tenaga kependidikan Departemen Ilmu Komputer IPB

DAFTAR PUSTAKA

Alimucaj A. 2009. Control Flow Graph Generator Documentation. [Internet]. [diunduh 2017 Juli 19]. Tersedia pada: http://eclipsefcg.sourceforge.net/ documentation.pdf.

Arkeman Y, Herdiyeni Y, Hermadi I, dan Laxmi G F. 2014. Algoritma Genetika Tujuan Jamak (MultiObjective Genetic Algorithm. Bogor (ID). IPB Press.

Basu A. 2015. Software Quality Assurance, Testing and Metrics. PHI Learning Privat Limited.

Dixon J. 2013. Graphviz.Net C# Wrapper. [Internet]. [diunduh 2017 Desember 22]. Tersedia pada: https://github.com/JamieDixon/GraphViz-C-Sharp-Wrapper.

Ellson J *et al.* 2003. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. [Internet]. [diunduh 2017 Agustus 14]. Tersedia pada: https://github.com/JamieDixon/ GraphVizC-Sharp-Wrapper.

Hermadi I. 2015. Path Testing using Genetic Algorithm. Disertasi. University of New South Wales.

Houcque D. 2015. Intoruction To MATLAB For Engineering Students. [Internet]. [diunduh 2017 Desember 30]. Tersedia pada: https://www.mccormick.northwestern.edu/documents /students/undergraduate/introductionto-matlab.pdf.

Kumar D dan Mishra K K. 2016. The Impacts of Test Automation on Software's Cost, Quality and Time to Market dalam: *Procedia Computer Science* 79,pp. 8–15. [Internet]. [diunduh 2017 Agustus 20]. Tersedia pada: http://www.sciencedirect.com/science/article/ pii/%5C%5CS18770509160012.

Myers G J, Sandler C, dan Badgett T. 2012. The Art of Software Testing. John Willey dan Sons, Inc, Hoboken, New York.

Suffos S. 2015. Matlab Parser. [Internet]. [diunduh 2017 Desember 22]. Tersedia pada: https//github.com/samuel-suffos/matlabparser.

Tikir M M dan Hollingsworth J K. 2011. Efficient Instrumentation for Code Coverage Testing dalam: *International Journal of Software Engineering and* Its Applications. [Internet]. [diunduh 2017 Agustus 21]. Tersedia pada: https://www.researchgate.net/publication/2835608_Efficient_Instrumentation_for_Code_Coverage_Testing.

Watson A H dan McCabe T J. 1996. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric) dalam: *NIST Special Publication*. [Internet]. [diunduh 2017 Agustus 14]. Tersedia pada: http://www.mccabe.com/pdf/mccabe-nist235r.pdf.