Instrumentasi Kode Program Secara Otomatis untuk *Basis Path Testing*

Raden Asri Ramadhina Fitriani(G64154007)*, Irman Hermadi

Abstrak/*Abstract*

Pengujian adalah serangkaian proses yang dirancang untuk memastikan sebuah perangkat lunak melakukan apa yang seharusnya dilakukan dan bertujuan untuk menemukan kesalahan pada perangkat lunak. *Basis Path testing* merupakan salah satu metode pengujian struktural yang menggunakan kode program untuk menemukan semua jalur yang mungkin dapat dilalui ketika program tersebut dijalankan dan dapat digunakan untuk merancang data uji. Untuk menguji perangkat lunak yang kompleks secara keseluruhan akan memakan waktu yang lama dan membutuhkan sumber daya manusia yang banyak. Idealnya, pengujian dilakukan untuk semua kemungkinan dari perangkat lunak. Kumar dan Mishra (2016) mengatakan bahwa pengujian perangkat lunak menggunakan hampir 60% dari total biaya pengembangan perangkat lunak. Sehingga mengotomasi bagian dari pengujian akan membuat proses ini menjadi lebih cepat dan mengurangi kerawanan akan kesalahan. Pada penelitian ini, akan 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. Untuk memonitor jalur yang dilalui program ketika dijalankan dengan masukan data uji tertentu, maka sistem ini juga akan melakukan instrumentasi kode program secara otomatis. Program yang akan diuji dalam penelitian ini adalah program yang dibangun dengan menggunakan bahasa Matlab. Dalam pengembangannya, aplikasi ini akan dibangun dengan menggunakan bahasa pemrograman C# dan *library* Graphiz.Net C# Wrapper untuk memvisualisasikan *Control Flow Graph*.

Kata Kunci

Basis Path Testing; *Control Flow Graph*; Instrumentasi

***Alamat Email**: radenasrirf@gmail.com

PENDAHULUAN

Latar Belakang

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. Saat pengujian, bisa saja tidak ditemukan kesalahan pada hasil pengujian. Hal ini dapat terjadi karena perangkat lunak yang sudah berkualitas tinggi atau karena proses pengujiannya berkualitas rendah. (Myers *et al.* 2012)

Teknik pengujian secara umum dibagi menjadi 2 kategori diantaranya *black box testing* dan *white box testing*. *Black box testing* bertujuan untuk memeriksa fungsional dari perangkat lunak apakah output sudah sesuai dengan yang ditentukan. Sedangkan *white box testing* atau biasa disebut dengan pengujian struktural merupakan pemeriksaan struktur dan alur logika suatu proses. *Basis 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). Untuk melakukan monitoring jalur mana yang diambil oleh sebuah masukan pada saat eksekusi program, maka diperlukan penanda yang dapat memberikan informasi cabang mana yang dilalui. 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. Tetapi 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. Jika proses pengujian perangkat lunak dapat dilakukan secara otomatis, maka hal ini dapat mengurangi biaya pengembangan secara signifikan.

Hermadi (2015) melakukan penelitian membangkitkan data uji untuk *path testing* menggunakan algoritma genetika. Dalam penelitian tersebut, Hermadi membangkitkan *Control Flow Graph* (CFG) dan instrumentasi masih secara manual sehingga membutuhkan banyak waktu dan rawan akan kesalahan ketika program sudah semakin besar. Sehingga mengotomasi hal tersebut dapat membuat *path testing* menjadi lebih cepat dan dapat mengurangi kerawanan akan kesalahan.

Pada penelitian ini, akan dibangun sebuah perangkat lunak untuk membangkitkan kemungkinan jalur dari

sebuah program. Jalur-jalur ini dapat dijadikan dasar untuk membangkitkan data uji agar data uji yang digunakan untuk pengujian dapat mewakili semua kemungkinan. Untuk memonitor jalur mana yang dilalui ketika diberikan masukan data uji, maka sistem ini juga akan melakukan penyisipan tag-tag sebagai instrumentasi ke dalam kode program secara otomatis.

Perumusan Masalah

Berdasarkan latar belakang di atas dapat dirumuskan masalahnya adalah bagaimana membangun sebuah aplikasi untuk melakukan instrumentasi secara otomatis untuk pengujian jalur dan dapat dimanfaatkan untuk *re-engineering* perangkat lunak.

Tujuan

Penelitian ini bertujuan untuk membangun sebuah aplikasi yang dapat digunakan untuk membangkitkan CFG dan melakukan instrumentasi secara otomatis.

Ruang Lingkup

Bahasa pemrograman yang diakomodasi adalah Matlab dan model diagram yang dibangkitkan adalah CFG.

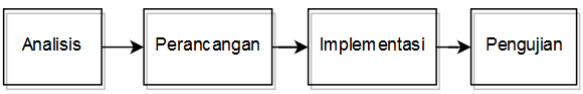
Manfaat

Hasil penelitian diharapkan dapat membantu pengembang dan penguji aplikasi untuk:

1. Menyisipkan tag-tag sebagai instrumentasi program ke dalam kode program secara otomatis sehingga proses tersebut dapat dilakukan dengan lebih cepat.
2. Membangkitkan jalur-jalur dasar yang dapat digunakan sebagai dasar untuk pembangkitan data uji.
3. Membangkitkan diagram CFG yang dapat memudahkan pengembang dalam memahami struktur dan alur dari suatu program yang dapat dimanfaatkan ketika akan melakukan *re-engineering* perangkat lunak.

METODE PENELITIAN

Penelitian yang dilakukan terbagi menjadi beberapa tahapan proses. Gambar 1 menunjukan tahapan proses tersebut.



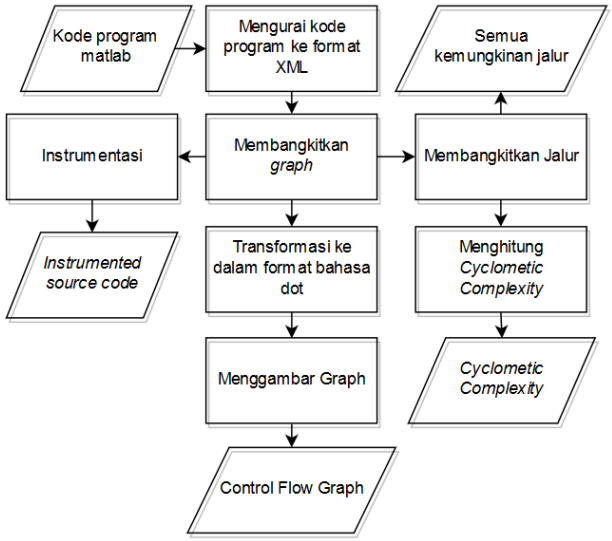
Gambar 1. Metode Penelitian

Analisis

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

Perancangan

Pada tahap ini ditentukan bagaimana perangkat lunak akan dibangun. Ilustrasi arsitektur sistem dapat dilihat pada Gambar 2.



Gambar 2. Arsitektur Sistem

Kode Program

Kode program matlab akan dibaca sebagai inputan. Matlab merupakan singkatan dari MATrix LABoratory. Seperti bahasa pemrograman lainnya, matlab memiliki beberapa sruktur kontrol. Struktur kontrol adalah perintah dalam bahasa pemrograman yang digunakan dalam pengambilan keputusan. Matlab memiliki empat struktur kontrol, yaitu IF-ELSE-END, SWITCH-CASE, FOR, dan WHILE (Houcque 2015).

Mengurai Kode Progam ke Format XML

Penguraian kode program matlab dilakukan dengan menggunakan library MATLAB-PARSER. Ketika terdapat kesalahan pada kode program, library ini akan mengembalikan pesan *error*. Lalu kode program tersebut diurai menjadi file dengan format XML menggunakan *library* MATLAB-PARSER yang dibuat oleh Suffos (2015).

Extensible Markup Language (XML) adalah bahasa yang dapat mendeskripsikan sebuah dokumen. XML

memiliki banyak bagian yang tidak memiliki struktur yang pasti. XML terdiri atas dua bagian utama, yaitu elemen dan atribut. Elemen yang dapat disebut sebagai *node* merupakan bagian penting yang dapat menggambarkan struktur dari XML. Sedangkan atribut merupakan bagian yang dapat digunakan sebegai informasi tambahan dari setiap elemen (Hartwell 2017).

Membangkitkan *Graph*

Setiap elemen dalam file XML tersebut akan ditelusuri satu persatu yang termasuk struktur kontrol di dalam bahasa matlab. Sehingga terbentuklah sebuah objek *graph* yang terdiri dari sekumpulan *node* dan *edge*.

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*. Setiap elemen XML yang merupakan struktur kontrol pada program akan menjadi *node* baru di dalam kelas *graph*. Setiap *node* berisi informasi nomor baris dan kolom yang akan digunakan untuk melakukan instrumentasi.

Membangkitkan Jalur

Basis path testing merupakan salah satu metode pengujian struktural yang menggunakan *source code* dari 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).

Metode ini terbagi menjadi 4 tahapan, yaitu:

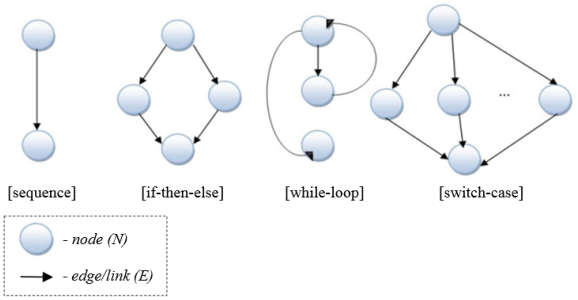
1. Menggambarkan jalur dalam bentuk *Control Flow Graph* (CFG)
2. Menghitung *cyclomatic complexity*
3. Memilih satu set jalur dasar
4. Membangkitkan data uji untuk setiap jalur dasar

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. Bahasa ini dapat mendeskripsikan 3 macam objek, yaitu *graph*, *nodes*, dan *edges* (Gansner *et al.* 2015).

Memvisualisasi *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 *statement* atau *expressions*. Sedangkan *edges* merepresentasikan transfer kontrol antar *nodes* (Watson dan McCabe 1996). Notasi dari CFG dapat dilihat pada Gambar 3. Setelah file dengan format bahasa dot terbentuk, CFG akan divisualisasikan dengan menggunakan library Graphviz. Graphviz merupakan perangkat lunak open source untuk visualisasi grafik. Graphviz memiliki banyak fitur berguna untuk menggambar diagram yang konkret karena terdapat pilihan warna, font, tata letak, jenis garis, dan bentuk (Ellson *et al.* 2003).



Gambar 3. Notasi Control Flow Graph (CFG)

Menghitung *Cyclometic Complexity*

Cyclomatic complexity merupakan suatu sistem pengukuran yang ditemukan oleh Watson dan McCabe 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 berikut:

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

Dimana, E menunjukkan jumlah *edges* dan N menunjukkan jumlah *nodes*.

Instrumentasi

Setelah jalur terbentuk, dilakukan juga proses instrumentasi. Instrumentasi merupakan sebuah proses menyisipkan sebuah penanda (tag) di awal atau di akhir setiap blok kode seperti awal setiap perintah, sebelum atau sesudah kondisi terpenuhi atau tidak. Dalam pengujian path testing, penanda ini dapat digunakan untuk memonitor jalur yang dilalui program ketika dijalankan dengan masukan data uji tertentu (Arkeman *et al.* 2014).

Instrumentasi dilakukan dengan cara menambahkan dulu variabel keluaran bernama *traversedPath*. Variabel in digunakan untuk menyimpan informasi node mana saja yang dilalui ketika diberikan inputan dengan nilai tertentu. Lalu 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.

Setelah file dengan format bahasa dot terbentuk, CFG divisualisasikan dengan menggunakan *library* Graphviz.Net. Graphviz.Net adalah pembungkus C# untuk generator grafik Graphviz yang dibuat oleh Dixon (2013). Keluaran yang dikembalikan ketika mengeksekusi Graphviz.Net berbentuk *byte* dalam *array* sehingga dapat diolah kembali sesuai dengan kebutuhan. Graphviz merupakan library yang dapat digunakan untuk menvisualisasi jalur ke dalam bentuk graph berarah (Gansner *et al.* 2015).

Testing

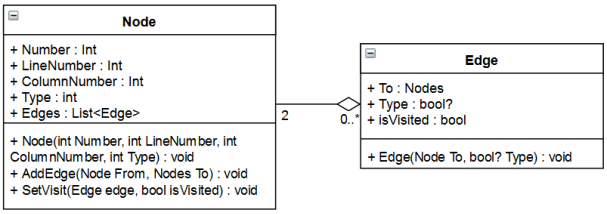
Tahapan ini adalah melakukan evaluasi dari tahapan implementasi. Evaluasi dibagi menjadi dua bagian, yaitu uji validasi dan uji efisiensi. Uji validasi dilakukan dengan cara membandingkan hasil yang ada pada penelitian sebelumnya dengan hasil yang dikeluarkan oleh aplikasi. Pada penelitian sebelumnya, *graph* yang dibangun adalah *graph* yang hanya menggambarkan notasi percabangan. 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.

Uji efisiensi dilakukan dengan membandingkan waktu eksekusi yang dilaukan secara manual dengan waktu eksekusi oleh aplikasi. Pengujian manual akan dilakukan dengan meminta satu atau dua orang yang sudah memiliki pengalaman dalam pemrograman sebagai sampel untuk melihat berapa lama waktu yang dibutuhkan untuk membangkitkan CFG, membangkitkan semua kemungkinan jalur, menghitung *cyclomatic complexity*, dan melakukan instrumentasi.

HASIL DAN PEMBAHASAN

Analisis

Pada penelitian ini, akan dibangun sebuah perangkat lunak untuk membangkitkan kemungkinan jalur dari sebuah program. Jalur-jalur ini dapat dijadikan dasar untuk membangkitkan data uji agar data uji yang digunakan untuk pengujian dapat mewakili semua kemungkinan. Untuk memonitor jalur mana yang dilalui ketika diberikan masukan data uji, maka sistem ini juga akan melakukan penyisipan tag-tag sebagai instrumentasi ke dalam kode program secara otomatis.



Gambar 4. *Perancangan Class Diagram*

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).

Data yang digunakan dalam penelitian ini didapatkan dari penelitian yang dilakukan oleh Hermadi (2015). Terdapat 15 contoh program yang akan digunakan pada penelitian ini dengan tingkat kompleksitas yang beragam. Contoh program yang akan digunakan dapat dilihat pada 1.

Perancangan

Perancangan *Class Diagram*

Class diagram dibangun untuk menggambarkan struktur sistem dari segi pendefinisian *class* dan huubungan antar *class*. Perancangan *class diagram* dapat dilihat pada Gambar 4.

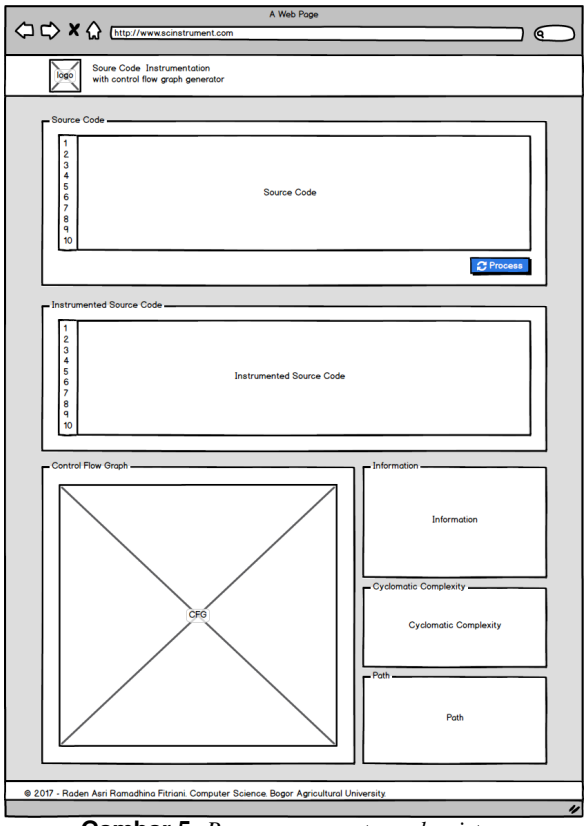
Dalam sebuah *class node* terdapat informasi nomor *node*, nomor baris dan nomor kolom dari kode program, dan tipe dari perintah tersebut apakah termasuk percabangan, pengulangan, perintah biasa, atau akhir dari sebuah perintah. Selain itu, terdapat list *edge* yang berisi *node* tujuan dan tipe dari *edge* yang digunakan jika terdapat percabangan *true*, *false*, atau hanya garis penghubung biasa.

Perancangan Antarmuka

Perancangan antarmuka meliputi perancangan antarmuka *form* untuk pengguna memasukkan kode program yang akan di proses dan antarmuka hasil dari proses yang telah dilakukan oleh aplikasi. Perancangan antarmuka hasil dari proses yang telah dilakukan dapat dilihat pada Gambar 5.

Tabel 1. Contoh program uji

No	Program Uji	Nama	Deskripsi
1	Triangle Ahmed	tA2008	Menentukan tipe dari segitiga apakah termasuk equilateral, isosceles, scalene, atau not triangle
2	Minimaxi Ahmed	mmA2008	Menentukannilai minimal dan maksimal dari inputan berupa bilangan dalam array
3	Binary Ahmed	binA2008	mencari indeks sebuah bilangan dalam array dengan mengembalikan indeks jika ditemukan dan tidak jika tidak ditemukan.
4	Bubble Ahmed	bubA2008	Mengurutkan bilangan dalam array menggunakan metode bubble sort
5	Quotient Bueno	qB2002	Menghitung hasil bagi dan sisa hasil bagi dari dua buah bilangan bulat positif
6	Fitness Minimaxi Hermadi	fmH2014	Menghitung fungsi fitness dari fungsi minimaxiAhmed2008
7	Insertion Ahmed	iA2008	Mengurutkan bilangan dalam array menggunakan metode insertion sort
8	Gcd Ahmed	gA2008	Menghitung GCD atau pembagi dua bilangan terbesar
9	Expint Bueno	eB2002	Fungsi exponensial yang dapat memproses bilangan integer dan float
10	Flex Gong	fG2011	Sebuah utilitas unix yang diambil dari situs GNU



Gambar 5. *Perancangan antarmuka sistem*

Implementasi

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

Sebagai contoh, kode program yang digunakan adalah tA2008. 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 6. *Kode program tA2008*

Kode Program

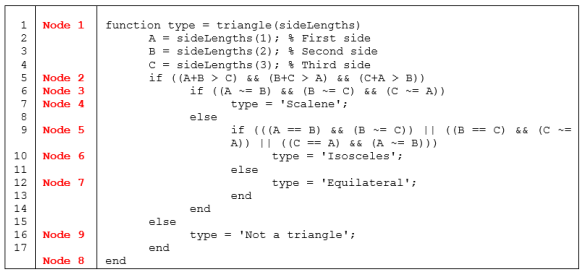
Gambar 6 merupakan kode program tA2008, yaitu fungsi untuk mencari jenis dari segitiga jika diketahui panjang dari setiap sisinya.

Mengurai Kode Progam ke Format XML

Penguraian kode program matlab dilakukan dengan menggunakan *library* MATLAB-PARSER. Kode program yang diinputkan harus sudah dipastikan dapat dijalankan jika di compile. Ketika terdapat kesalahan pada kode program, library ini akan mengembalikan pesan error. 7 menunjukkan potongan hasil penguraian kode program ke dalam format XML. Potongan kode XML yang terlihat pada 7 menunjukkan hasil penguraian dari kode program pada baris ke 6 sampai baris ke 7.

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. State-

Gambar 7. Potongan hasil penguraian kode program tA2008 ke dalam format XML

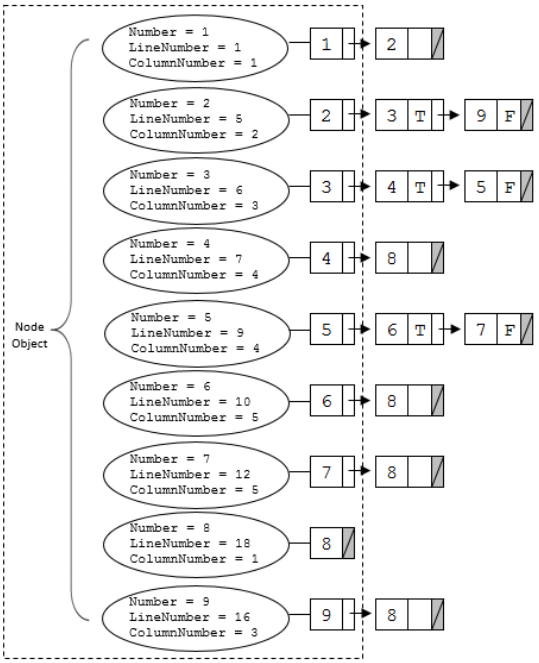


Gambar 8. Hasil nodes yang terbentuk dari tA2008

ments¿. Sedangkan untuk bagian yang tidak memenuhi kondisi IF atau bagian ELSE akan disimpan di dalam elemen ¡If.ElsePart¿¡/If.ElsePart¿.

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*. Setiap elemen XML yang merupakan struktur kontrol pada program akan menjadi *node* baru di dalam kelas *graph*. Setiap *node* berisi informasi nomor baris dan kolom yang akan digunakan untuk melakukan instrumentasi. Setiap *node* juga dapat memiliki *edge* yang berisi informasi *node* tujuan dan tipe dari garis penghubung itu sendiri. Terdapat tiga macam tipe pada *edge* yaitu *null*, *true*, dan *false*. *True* dan *false* digunakan jika *node* asal merupakan percabangan. Hasil *node* yang dibentuk dari kode program tA2008 dapat dilihat pada 8. *Node* 1 dibentuk pada awal kode program sebagai inisialisasi. *Node* ditambahkan ketika bertemu dengan perintah yang termasuk ke dalam struktur kontrol seperti IF-ELSE-END, SWITCH-CASE, FOR, dan WHILE. Seperti dapat dilihat pada baris kode ke 5, terdapat perintah IF sehingga dibentuk *node* baru yaitu *node* 2.



Gambar 9. Object diagram tA2008

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

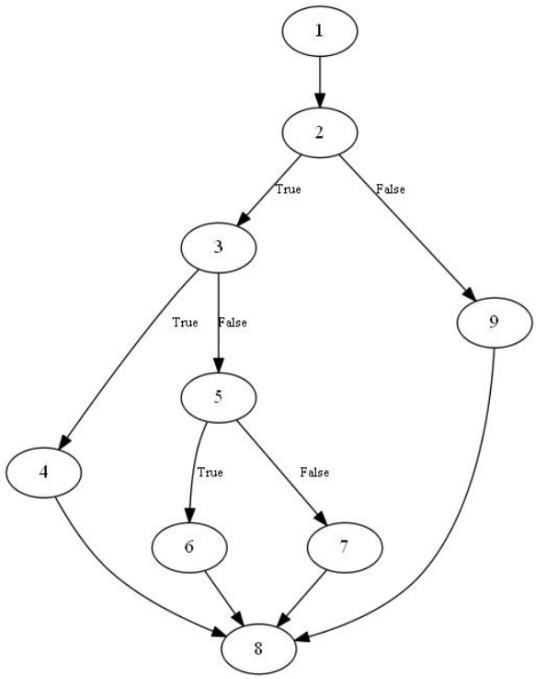
Membangkitkan Jalur

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. (T) ketika melalui *edge* yang memiliki tipe *true*, dan (F) ketika melalui *edge* yang memiliki tipe *false*. Setiap *edge* memiliki atribut *isVisited* yang digunakan untuk menandai apakah garis penghubung tersebut sudah dilalui atau belum. Jalur yang dibentuk ketika melalui perintah pengulangan seperti FOR dan WHILE akan dibatasi hanya satu kali pengulangan. Berikut merupakan semua kemungkinan jalur yang akan dilalui ketika diberikan suatu inputan yang dapat dijadikan sebagai dasar dalam pembangkitan data uji.

1. 1 2 (T) 3 (T) 4 8
2. 1 2 (T) 3 (F) 5 (T) 6 8
3. 1 2 (T) 3 (F) 5 (F) 7 8
4. 1 2 (F) 9 8

```
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 10. Representasi tA2008 dalam bahasa dot



Gambar 11. CFG tA2008

Transformasi ke Dalam Format Bahasa Dot

Transformasi ke dalam format bahasa dot dilakukan dengan cara menelusuri objek *graph* yang sudah dibangun sebelumnya. Yang didefinisikan dalam bahasa dot adalah *edge* yang terdapat pada *graph* yang dibangun. Seperti yang dapat dilihat pada 10, jumlah baris sebanyak jumlah *edge* pada objek *graph* yang telah didefinisikan sebelumnya.

Memvisualisasi *Graph* dalam bentuk CFG

Setelah file dengan format bahasa dot terbentuk, CFG divisualisasikan dengan menggunakan *library* Graphviz.Net. Hasil visualisasi bahasa dot kode program tA2008 ke dalam CFG dapat dilihat pada 11.

Menghitung *Cyclometic Complexity*

Cyclomatic complexity merupakan suatu sistem pengukuran yang menunjukkan banyaknya *independent path*. *Cyclomatic Complexity* dihitung dengan cara jumlah *edge* dikurangi dengan jumlah *node*, lalu ditambahkan dengan dua. Berdasarkan *graph* yang telah terbentuk dari kode program tA2008, dapat dilihat pada 9 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.

$$Nodes(N) = 9$$

$$Edges(E) = 11$$

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

$$= 4$$

Instrumentasi

Instrumentasi dilakukan dengan cara menambahkan dulu variabel keluaran bernama traversedPath. Variabel in digunakan untuk menyimpan informasi *node* mana saja yang dilalui ketika diberikan inputan dengan nilai tertentu. Dan menyimpan informasi pilihan yang dilalui ketika ditemukan cabang yang terdapat pilihan *true* atau *false*.

Hasil kode program yang telah diinstrumentasi dapat dilihat pada 12. Sebelumnya, kode program tA2008 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 12.

Sehingga ketika kode program hasil instrumentasi dijalankan dengan inputan tertentu akan menghasilkan keluaran variabel *traversedPath* dan *type* seperti yang dapat dilihat pada Gambar 13. Misalkan inputan adalah 3, 4, dan 4 akan menghasilkan *isosceles* dan dapat diketahui bagaimana cara menghasilkan keluaran tersebut dari *traversedPath*.

Implementasi Antarmuka

Tampilan hasil pembangkitan dapat dilihat pada Gambar 14.

Testing

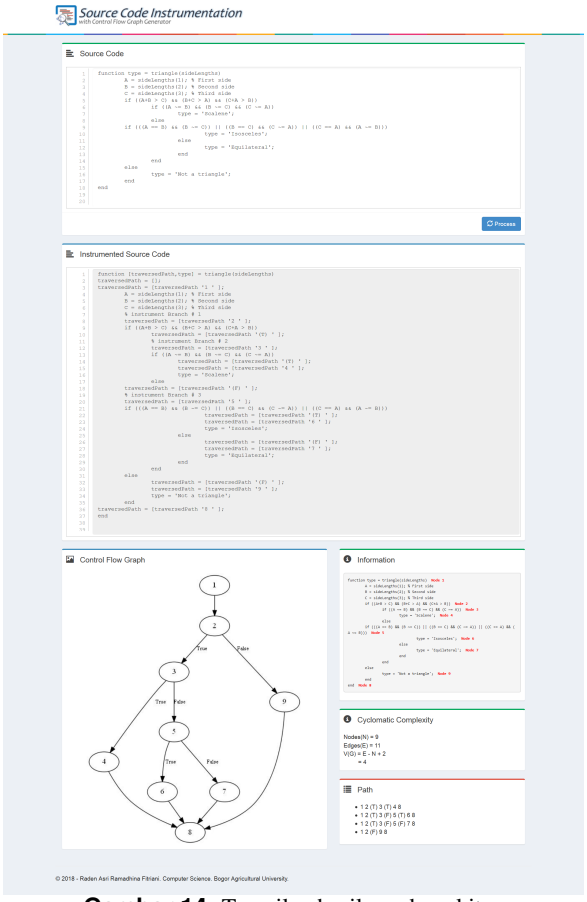
Tahapan ini adalah melakukan evaluasi dari tahapan implementasi. Evaluasi dibagi menjadi dua bagian, yaitu uji validasi dan uji efisiensi. Uji validasi dilakukan dengan cara membandingkan hasil yang ada pada penelitian sebelumnya dengan hasil yang dikeluarkan oleh

```
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 12. Hasil instrumentasi tA2008

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

traversedPath =

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


type =

    'Isosceles'
```

Gambar 13. Hasil ekseskusi kode program tA2008 yang sudah diinstrumentasi



Gambar 14. Tampilan hasil pembangkitan

aplikasi. Pada penelitian sebelumnya, *graph* yang dibangun adalah *graph* yang hanya menggambarkan notasi percabangan. 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. Tabel 2 menunjukkan 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. Seperti pada contoh program tA2008, *node* 4, 5, 6, 7, 8, 9 menjadi *node* 9, 4, 5, 6, 7, 8 di *graph* yang dibangun menggunakan aplikasi.

Uji efisiensi dilakukan dengan membandingkan waktu eksekusi yang dilaukan secara manual dengan waktu eksekusi oleh aplikasi. Penguji terdiri dari dua orang yang berprofesi sebagai pengembang sistem. Hasil perbandingan waktu yang dibutuhkan untuk melakukan pembangkitan secara manual dan oleh aplikasi dapat dilihat pada Tabel 3. Program uji nomor 1 tA2008 tidak ada waktu eksekusi secara manual karena program tersebut sudah digunakan sebagai contoh.

Waktu yang dibutuhkan aplikasi untuk membangkitkan CFG, melakukan instrumentasi, menghitung *cyclomatic complexity*, dan membangkitkan semua kemungkinan jalur rata-rata 1.64 detik. Sedangkan jika hal tersebut dilakukan secara manual, akan menghabiskan waktu rata-rata 383.28 detik atau 6 menit 23 detik. Jumlah waktu yang dibutuhkan juga akan semakin meningkat ketika kode program semakin kompleks seperti pada program uji nomor 9 eB2002 yang dapat menghabiskan waktu rata-rata 659.43 detik atau 10 menit 59 detik. Sedangkan tidak berpengaruh ketika dieksekusi menggunakan aplikasi yang hanya menghabiskan waktu 1.47 detik.

KESIMPULAN DAN SARAN

Kesimpulan

Penelitian ini berhasil membangun sebuah aplikasi yang dapat digunakan untuk melakukan instrumentasi secara otomatis, membangkitkan CFG, menghitung cyclomatic

Tabel 2. Perbandingan *adjacency list* manual dan menggunakan aplikasi

No	Nama Program	Adjacency List Penelitian Sebelumnya	Adjacency List Menggunakan Aplikasi
1	tA2008	· 1 ->2 · 2 ->3 ->4 · 3 ->5 ->6 · 4 ->9 · 5 ->9 · 6 ->7 ->8 · 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	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
4	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
5	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
6	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
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	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
9	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
10	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

Tabel 3. Perbandingan waktu eksekusi secara menual dan menggunakan aplikasi

No	Nama Program	Waktu Eksekusi Aplikasi (detik)	Waktu Ekseskusi Manual (detik)		
			Penguji 1	Penguji 2	Rata-Rata
1	tA2008	1.53	-	-	-
2	mmA2008	1.71	558.23	407.90	483.07
3	binA2008	1.45	384.64	526.59	455.62
4	bubA2008	1.39	448.64	207.97	328.31
5	qB2002	1.75	222.25	280.86	251.56
6	fmH2014	1.81	201.87	223.07	212.47
7	iA2008	1.56	234.55	358.51	296.53
8	gA2008	2.09	492.93	293.01	392.97
9	eB2002	1.47	686.59	632.27	659.43
10	fG2011	1.64	335.92	403.27	369.60
	Rata-Rata	**1.64**	396.18	370.38	**383.28**

complexity, dan membangkitkan segala kemungkinan jalur yang dapat dilewati dari kode program matlab.

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. Selain itu, instrumentasi juga dapat dilakukan dinamis sesuai dengan kondisi yang dibutuhkan. Seperti yang dilakukan oleh Hermadi, selain menyimpan informasi jalur mana yang dilewati, instrumentasi yang dilakukan pada penelitian tersebut juga menyisipkan kode program untuk menghitung nilai *fitness*.

DAFTAR PUSTAKA

Alimucaj, Aldi. 2009. "Control Flow Graph Generator Documentation". [Internet]. [Diunduh tanggal 19/7/2017]. Dapat diunduh dari: http://eclipsefcg.sourceforge.net/Documentation.pdf.

Arkeman, Y, Herdiyeni, Y, Hermadi, I, dan Laxmi, G F. 2014. *Algoritma Genetika Tujuan Jamak (Multi-Objective Genetic Algorithm*. IPB Press.

Basu, A. 2015. *Software Quality Assurance, Testing and Metrics*. PHI Learning Privat Limited. [Internet]. [Diunduh tanggal 14/8/2017]. Dapat diunduh dari: https://books.google.co.id/books.

Dixon, J. 2013. "Graphviz.Net C# Wrapper". [Internet]. [Diunduh tanggal 22/12/2017]. Dapat diunduh dari: https://github.com/JamieDixon/GraphViz-C-Sharp-Wrapper.

Ellson, J *et al.* 2003. "Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools". [Internet]. [Diunduh tanggal 14/8/2017]. Dapat diunduh dari: https://github.com/JamieDixon/GraphViz-C-Sharp-Wrapper.

Gansner, Emden R., Koutsofios, Eleftherios, dan North, Stephen. 2015. "Drawing graphs with dot". [Internet]. [Diunduh tanggal 25/12/2017]. Dapat diunduh dari: www.graphviz.org/pdf/dotguide.pdf.

Hartwell, J. 2017. *C# and XML Primer*. Apress.

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

Houcque, David. 2015. "Intoruction To MATLAB For Engineering Students". [Internet]. [Diunduh tanggal 30/12/2017]. Dapat diunduh dari: https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-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 tanggal 20/8/2017]. Dapat diunduh dari: http://www.sciencedirect.com/science/article/pii/%5C%5CS1877050916001277.

Myers, G J, Sandler, C, dan Badgett, T. 2012. *The Art of Software Testing*. John Willey dan Sons, Inc, Hoboken, New York. [Internet]. [Diunduh tanggal 14/8/2017]. Dapat diunduh dari: https://books.google.co.id/books.

Suffos, S. 2015. "Matlab Parser". [Internet]. [Diunduh tanggal 22/12/2017]. Dapat diunduh dari: https://github.com/samuel-suffos/matlab-parser.

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 tanggal 21/8/2017]. Dapat diunduh dari: 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 tanggal 14/8/2017]. Dapat diunduh dari: http://www.mccabe.com/pdf/mccabe-nist235r.pdf.