Evaluasi Kemampuan Tool TRGeneration terhadap Variasi Logical Complexity Program

Asri Maspupah[1], Ani Rahmani[2], Joe Lian Min[3]

[1,2,3]*Jurusan Teknik Komputer dan Informatika, Politeknik Negeri Bandung, Bandung 40012*
E-mail : asri.maspupah@polban.ac.id, anirahma@jtk.polban.ac.id, joelianmin@jtk.polban.ac.id

ABSTRAK

Software testing merupakan kegiatan untuk mengevaluasi *software* yang dikembangkan agar dicapai kualitas tertentu. Salah satu bagian penting pada pengujian *software* adalah identifikasi kasus uji untuk dapat dibuat *independent path* unit. Terdapat banyak *tool* untuk mengidentifikasi *independent path*. Masalahnya, tool yang ada belum tentu mampu menangani seluruh jenis struktur (variasi *logical complexity)* dari suatu program. Tulisan ini mengangkat hasil evaluasi kemampuan TRGeneration -- sebuah *open source* berbahasa Java -- yang memanfaatkan *control flow graph* (CFG) untuk membuat *independent path.* TRGeneration dipilih karena memiliki kelebihan dalam melakukan visualisasi CFG dari sebuah *source code*. Visualisasi CFG diperlukan oleh *tester* untuk membantu melengkapi *test case*. Variasi struktur program yang dievaluasi terdiri atas 4 bentuk struktur, yaitu *sequence*, *selection* dan *repetition*, serta kombinasi dari ketiga struktur tersebut, dengan total variasi yang diamati 41 jenis. Evaluasi dilakukan melalui pengamatan terhadap setiap jenis struktur program, dengan melihat kesesuaian kompleksitas dengan luaran berupa CFG dan *independent path*. *Independent path* yang dihasilkan dievaluasi menggunakan *code coverage criteria*, yaitu pendekatan untuk memeriksa kelengkapan *test case* dari proses pengujian unit. Dari hasil pengamatan diketahui bahwa dari 41 variasi, baru 31 yang dapat ditangani. Artinya, TRGeneration belum dapat digunakan untuk pengujian, dan masih perlu disempurnakan agar dapat menangani keseluruhan variasi *logical complexity*.

Kata Kunci
Software testing, control flow graph, independent path, TRGeneration, logical complexity

1. PENDAHULUAN

Salah satu pendekatan pengujian software dapat dilakukan pada pengujian unit (*unit testing*). *Unit testing* berfokus pada logika pemrosesan ruang lingkup unit untuk memeriksa setiap *statement* program dan struktur program apakah sudah sesuai dengan fungsinya [2]. Pelaksanaan pengujian unit dilakukan dengan mengidentifikasi *item test case*, kemudian evaluasi *source code* berdasarkan *test case* tersebut. Namun permasalahan selanjutnya adalah apakah semua *item test case* yang disiapkan sudah meliputi seluruh *statement* program.

Pendekatan yang dapat dilakukan untuk mengukur kelengkapan *test case* pada pengujian unit adalah analisa *code coverage*. Sebelum melakukan analisa *code coverage* diperlukan representasi data yang menggambarkan alur *logic program* sehingga dapat diidentifikasi seluruh *item test case* pengujian yang diperlukan. *Control flow graph* (CFG) merupakan salah satu metode yang digunakan untuk menggambarkan *alur logic program* [6].

Saat ini terdapat banyak *tools* untuk membentuk CFG dan mengidentifikasi item pengujian yang memungkinkan dari *alur logic program*, namun tool yang ada belum tentu mampu menangani seluruh jenis struktur dan *logical complexity* dari suatu program. Salah satu *tool* untuk membentuk CFG dan mengidentifikasi item pengujian adalah TRGeneration. TRGeneration dipilih karena mampu melakukan visualisasi CFG ke dalam bentuk *graph*. Visualisasi CFG penting untuk membantu *tester* dalam upaya melengkapi *test case* pada proses pengujian unit.

Penelitian yang dilakukan adalah menelaah kemampuan TRGeneration dalam membentuk CFG dan memvisualisasikannya ke dalam bentuk *graph* serta mengidentifikasi jalur *logic program*. Aktivitas untuk mengevaluasi adalah dengan melakukan eksplorasi dan observasi terhadap TRGeneration dengan mencoba berbagai variasi *logical complexity*. Hasil akhir penelitian berupa pengetahuan berapa *logical complexity* yang dapat ditangani oleh TRGeneration dan bagaimana strategi yang mungkin dapat dilakukan untuk menyempurnakan TRGeneration agar dapat dimanfaatkan dalam proses pengujian terutama dalam menganalisis kelengkapan *test case*.

2. PENELITIAN TERKAIT

2.1. Strategi Evaluasi Code Coverage Tool

Study comparative mengenai evaluasi berbagai *code coverage tool* dapat dilakukan dengan berbagai macam strategi. Salah satu strateginya adalah melalui pengamatan kinerja dan kemampuan tool yang dipandang terhadap aspek *code coverage*. Pengamatan seperti yang dilakukan Sneha pada tahun 2014, yaitu dengan melakukan *literature review*, *program instrument* dan analisa penerapan *coverage measurement criteria* terhadap performa tool [8]. *Program instrumen* adalah proses penangkapan informasi *code*

coverage melalui *monitoring program execution*. *Monitoring program execution* dilakukan dengan memasukan beberapa baris *source code* sebelum *code coverage tool* dieksusi. Kemudian setelah program dieksekusi, *tool* menghasilkan catatan pelaksanaan program. Selanjutnya diamati kemampuan *code coverage* beradasarkan *coverage measurement criteria*, yaitu *statement coverage*, *decision coverage*, *bloc <k coverage* dan *function/method coverage*. Hasil akhir penelitian Sneha adalah ringkasan kemampuan *code coverage tool* terhadap 5 jenis tool berdasarkan aspek bahasa pemograman *source code* yang ditangani tool, *coverage measurement criteria*, dan informasi tentang sudah mampu melakukan *automation testing* atau tidak.

Berdasarkan strategi pendekatan di atas, tulisan ini mengangkat pendekatan evaluasi yang serupa. Strategi *evaluasi code coverage tool* dilakukan dengan mengobservasi kemampuan TRGeneration untuk membentuk CFG, kemudian memvisualisasikannya ke dalam bentuk *graph* dan pembentukan *independent path*. Bagian yang diadopsi adalah penambahan varisi *source code* untuk mengamati pembentukan CFG.

2.2. TRGeneration

TRGeneration sebuah *open source* dalam bahasa java yang dibuat oleh Seung Hun (Stan) Lee dan Evan Platt untuk melakukan visualisasi CFG yang digunakan pada pengujian. Visualiasi CFG menggambarkan *logical complexity* dari sebuah *program* dalam bentuk *control flow graph*. Terdapat beberapa kemampuan yang dapat ditangani oleh TRGeneration [3]:

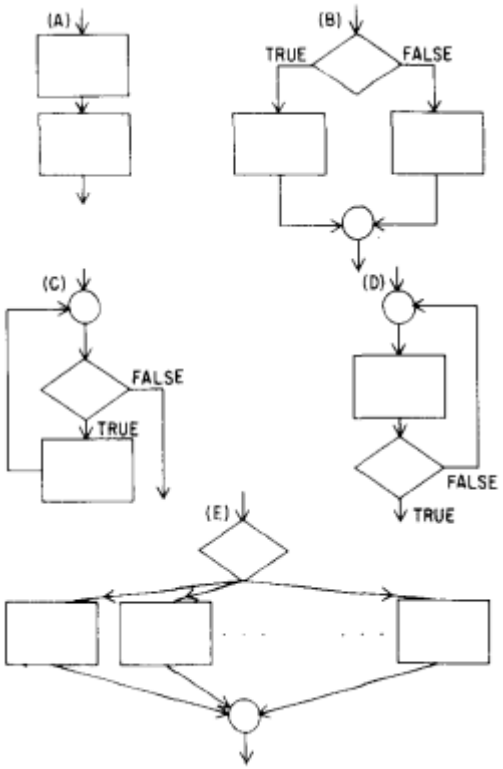
a. Sebagai *graph generator*, yaitu kemampuan TRGeneration yang bertanggung jawab untuk melakukan visualisasi CFG yang dibentuk dari struktur *program*. TRGeneration membaca *source code* program dalam berbagai bahasa pemrograman yang mengacu pada kaidah penulisan bahasa C sebagai *mother languange*. Kemudian, TRGeneration menguraikan *program* ke dalam bentuk CFG yang dibuat dari *node dan edge*. Penguraian program dilakukan dengan *parsing* sintak program berdasarkan *logical complexity* yang mencangkup struktur *sequence*, *selection* dan *repetition*.

b. Sebagai *test requirement generator*, yaitu kemampuan TRGeneration yang bertanggung jawab untuk membentuk *independent path* berdasarkan *code coverage criteria*. TRGeneration membaca *CFG* kemudian mengidentifikasi *coverage criteria*. Namun hanya 3 dari 4 *coverage criteria* yang mampu ditangani oleh TRGeneration yaitu *statement coverage*, *branch coverage* dan *path coverage*. Sementara *conditional coverage* belum ditangani.

Visualisasi CFG sebagai output dari TRGeneration dapat memberikan pemahaman kepada tester dalam melihat *logical compelexity* program yang sedang diuji sebagai alat bantu indentifikasi kasus uji. Namun perlu memastikan kemampuan TRGeneration dalam memvisualisasikan semua jenis struktur program.

2.2 Logical Complexity Program

Logical complexity dari suatu program merepresentasikan tingkat kerumitan struktur program yang digambarkan dalam bentuk diagram CFG. *Control flow program* adalah aliran proses jalur kendali *logic program* dari suatu *data input* sampai menghasilkan *output program*. Pada *structure programming*, *CFG* terdiri dari tiga struktur pemrograman yaitu *sequence*, *selection* dan *repetition* [4] [5].
Struktur *sequence* merepresentasian *statement* program yang memiliki eksekusi alur program yang berurutan. Struktur *selection* merepresentasikan eksekusi pemilihan alur program berdasarkan kondisi true atau false. *Statement* program yang termasuk kedalam *selection* adalah IF-THEN (eksekusi *statement* program yang dilaksanakan jika kondisi true), IF-THEN-ELSE (eksekusi pemilihan *statement* program yang dilaksanakan berdasarkan kondisi *true* dan *false*), CASE (eksekusi pemilihan *statement* program dalam berbagai alternatif). Struktur *repetition* merepresentasikan pengulangan *statement* program selama kondisi bernilai *true*. *Statement* program yang termasuk ke dalam *repetition* adalah While-Do (eksekusi *statement* program yang dilaksanakan berulang selama kondisi *true*), Do-While (eksekusi *statement* program pengulangan yang dilaksanakan minimal satu kali selama kondisi *true*), Do-Until (eksekusi *statement* program pengulangan yang dilaksanakan minimal satu kali selama kondisi *false*) dan For (eksekusi *statement* program pengulangan yang sudah diketahui jumlah iterasi pengulangannya). Dengan demikian, struktur *selection* dan *looping* memiliki pencabangan alur program berdasarkan alur eksekusi pemilihan kondisi. Ilutrasi CFG ditunjukkan Gambar 1.

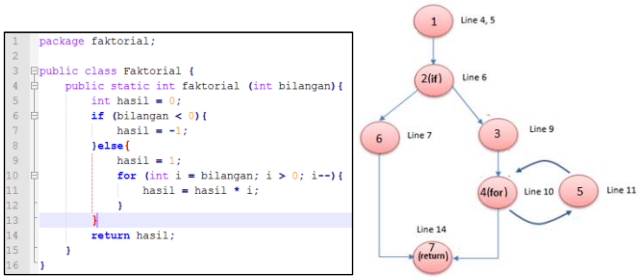


Gambar 1. Control Structure Program (A) Sequence (B) Selection If-Then-Else, (C) Repetition Do-While (D) Repetition Do-Until (E) Selection CASE [4]

2.3 Control Flow Graph

Dalam upaya memberi kemudahan bagi *tester*, informasi mengenai kelengkapan *test case* yang dirancang, digambarkan melalui visualiasi menggunakan *flow graph*. *Flow graph* menggambarkan *logical control flow* dengan notasi yang diilustrasikan pada Gambar 1 [2]. *Control flow graph* (CFG) pada tahap pengujian digunakan dalam pengidentifikasi kasus uji sebagai item *test case*. Hal ini karena CFG menggambarkan *logical control flow* [2] *Logical control flow* direpresentasikan sebagai *independent path* program [6]. *Independent path* menggambarkan satu jalur proses program dari *input* sampai membentuk *output*. Representasi CFG divisualisasikan dengan menggunakan *structure flow* yang ditulis dengan notasi *flow chart*.

Pembuatan CFG mengacu pada struktur pemrograman: *sequence*, *selection*, *repetition* dan bagian titik temu pada pencabangan dari struktur *selection* atau *repetition* [6]. Notasi CFG menggunakan simbol *node* dan *edge*. *Node* merepresentasikan satu atau lebih *procedural statement* program. Sedangkan *edge* merepresentasikan *control flow* atau aliran dari satu *node* menuju *node* lainnya.



Gambar 2. Control Flow Graph (CFG)

Gambar 2 menunjukkan CFG dari *program* factorial. Berdasarkan bentuk CFG di atas terbentuk 3 *independent path* yaitu Path 1: 1-2-6-7, Path 2: 1-2-3-7, dan Path 3: 1-2-3-4-5-4-7. Selanjutnya *independent path* menjadi *item test case* pada pengujian unit.

2.4 Code Coverage

Sebuah proses pengujian, harus dijamin lengkap dan tidak ada *path* atau alur program yang terlewat. Oleh karena itu, diperlukan alat ukur untuk memeriksa derajat kelengkapan *test case* pada sebuah *source code*. *Code coverage* adalah salah satu alat ukur untuk pengukuran derajat jumlah *statement* program yang telah dieksekusi [1]. Hal ini menyatakan presentase jumlah baris program yang telah diuji berdasarkan *data test* pada *item coverage*. Presentase tersebut bertujuan untuk mengidentifikasi informasi jumlah *independent path* yang tidak tercakup pada *test case* berdasarkan kriteria *coverage* tertentu. Analisa *code coverage* terhadap suatu program mengacu kepada *code coverage criteria* yang diadopsi dari konsep yang diperkenalkan oleh Arapdis pada tahun 2012, yaitu

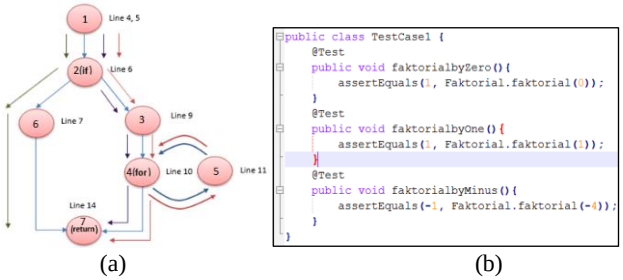
(1) *Condition coverage*, evaluasi terhadap *boolean expression* yang menghasilkan kondisi *true* atau *false*;

(2) *Branch coverage*, menjangkau semua *branch* yang berbeda pada *control flow structure code*.

(3) *Statement coverage*, semua *statement* dalam suatu *method* atau *block* tercakup pada *test case*.

(4) *Path coverage*, evaluasi semua *independent path* tercakup pada *test case*.

Contoh ilustrasi, penerapan analisis *code coverage* pada program factorial (Gambar 2) dapat dilihat pada Gambar 3.



(a) (b)

Gambar 3. Penerapan Code Coverage, (a) Analisis Code Coverage; (b) Test Case

Gambar 3 menunjukkan penerapan analisis *code coverage* pada *program* factorial menggunakan CFG berdasarkan tiga *item test case*. Hasil analisa pada Gambar 3.a menunjukkan bahwa *test case* telah memenuhi *code coverage citeria*

karena *independent path* sudah semuanya tereksekusi pada pengujian yang di-*trigger* oleh *data test*. *Independent path* diilustrasikan dengan garis panah berwarna hijau (Path 1), ungu (Path 2) dan merah (Path 3). Sementara Gambar 2.b memperlihatkan *data test* yang digunakan, yaitu 1) bilangan nol sebagai *data test* untuk eksekusi Path 2; 2) bilangan satu sebagai *data test* untuk eksekusi Path 3; dan 3) bilangan minus sebagai *data test* untuk eksekusi Path 1.

3. METODE

Evaluasi dilakukan dengan cara mengeksplorasi dan mengobservasi TRGeneration untuk melihat kemampuannya dalam hal pembentukan CFG terhadap *logical complexity* program. Eksplorasi dilakukan melalui uji coba TRGeneration dengan memasukkan berbagai *source code* program yang memiliki variasi CFG, kemudian dilakukan analisis terhadap hasil keluaran TRGeneration. Tahapan detail dari kegiatan observasi TRGeneraion dijelaskan adalah:

a. Mengidentifikasi variasi struktur *program* sebagai kasus uji observasi TRGeneration. Proses identifikasi struktur program dibuat berdasarkan *logical complexity* yang mengacu pada struktur pemrograman: *sequence*, *selection*, dan *repetition*. Tabel 1 memperlihatkan jenis variasi struktur program. Variasi didapatkan dengan melakukan pengamatan struktur program berdasarkan *sample* program yang ada di industri untuk melihat tingkat kerumitan struktur program. Kemudian hasil pemngamatan tersebut dikelompokkan berdasarkan struktur pemograman.

b. Mengobservasi kemampuan TRGeneration, aktifitas evaluasi hasil keluaran TRGeneration, CFG dan *independent path*, berdasarkan variasi *logical complexity* dari *source code* yang dimasukkan. CFG diperiksa dengan mengamati *source code* pada *node* dan *edge* yang terbentuk; keterhubungan antar *node* dan visualisasi CFG. Pengamatan evaluasi CFG dilakukan berdasarkan *logic control program*, sedangkan *independent path* diperiksa dengan mengikuti panduan dari *code coverage criteria*, yaitu memeriksa semua kemungkinan jalur/path eksekusi sudah semua teridentifikasi.

c. Mengobservasi kemampuan dan kelemahan TRGeneration berdasarkan uji kehandalan. Artinya, aktifitas menelaah kemampuan TRGeneration dari sejumlah variasi *logical complexity* berdasarkan struktur program yang ditangani dan yang tidak ditangani.

d. Melakukan *debugging* untuk melihat bagian yang *error*, karena tidak sesuai dengan pembentukan CFG.

e. Menyimpulkan jumlah presentase variasi struktur logical complexity yang dapat ditangani oleh TRgeneration.

5. EVALUASI HASIL DAN PEMBAHASAN

Evaluasi observasi kemampuan TRGeneration menyatakan seberapa handal TRGeneration dalam melakukan *graph generator* dan *test requirement generator*. Dari hasil pengamatan variasi struktur *program*, teridentifikasi 41 jenis struktur program dengan 4 kategori, yaitu: *sequence* (4 jenis struktur), *selection* (16 jenis struktur), *repetition* (12 jenis struktur), dan kombinasi dari ketiganya (9 jenis struktur). Data variasi struktur program ditunjukkan pada Tabel 1.

Hasil evaluasi pada Tabel 1 merepresentasikan status variasi struktur yang dapat ditangani oleh TRGeneration atau tidak. Hasil evaluasi yang menyatakan "V" menunjukkan bahwa CFG sudah benar dan *independent path* sesuai dengan *coverage criteria*, sedangkan hasil evaluasi yang menyatakan "X" menunjukkan bahwa CFG dan *independent path* tidak sesuai.

Tabel 1. Hasil Observasi Kemampuan Tool TRGeneration

No	Objektif Evaluasi terhadap Varian Logical Complexity Program	Hasil evaluasi	Keterangan
Struktur Sequence			
1.	Kasus komputasi ekspresi aritmatika sederhana, yaitu menggunakan operator: +, - dan bilangan numerik sebagai operand.	V	-
2.	Kasus komputasi ekspresi aritmatika cukup rumit yaitu menggunakan operator: +, -, *, /, (,), % dan bilangan numerik sebagai operand.	V	-
3.	Kasus komputasi ekspresi relasional dengan satu buah kondisi. Operator yang digunakan adalah >, <, !=, >=, <=, dan == sebagai ekspresi relasional perbandingan bilangan numerik. Operator != dan == dapat digunakan untuk perbandingan karakter/string.	V	-
4.	Kasus komputasi ekspresi penggabungan beberapa kondisi relasional pada perbandingan bilangan numerik atau karakter/string. Operator yang digunakan adalah >, <, !=, >=, <=, ==, and, or, !.	V	-
Struktur Selection			
5.	Struktur IF then EndIF.	V	-
6.	Struktur IF then-else EndIF.	V	-
7	Struktur IF then-else if-else EndIF dengan menggunakan variable yang memiliki tipe data int atau float.	V	-

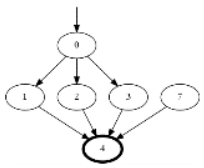
No	Objektif Evaluasi terhadap Varian Logical Complexity Program	Hasil evaluasi	Keterangan
8	Struktur IF then-else if-else EndIF dengan menggunakan variable yang memiliki tipe data double.	X	*CFG* tidak sesuai karena graph tidak terjadi percabangan

9.	Struktur *nested if* pada kasus IF then-else-[IF then-else EndIF] EndIF, contoh kode program:
	`if ( kondisi ) {`
	`  statement kode program`
	`}else {`
	`  if ( kondisi ) {`
	`  statement kode program`
	`  } else {`
	`  statement kode program`
	`  }`
	`}`
		X	*Graph* tidak terbentuk dan muncul pesan:
			"<terminate> TRGenerate[
			Else without if"

10.	Struktur *nested if* pada kasus IF then [IF then EndIF] EndIF	V	-

11.	Pada kasus 2 atau lebih struktur *selection* (misalnya IF then-else EndIF) yang ditulis berurutan setelah *end condition* pertama dilanjutkan dengan if berikutnya. contoh kode program:
	`if ( kondisi ) {`
	`  statement kode program`
	`}else if ( kondisi ) {`
	`  statement kode program`
	`}else{`
	`  statement kode program`
	`}`
	`if ( kondisi ) {`
	`  statement kode program`
	`}else {`
	`  statement kode program`
	`}`
		V	-

13.	Struktur Switch Case yang memiliki default pada akhir kondisi case	X	Penggambaran *CFG* dan *independent path* sudah sesuai. Namun nilai *case* tidak disimpan dalam *node* sehingga *graph* masih salah.

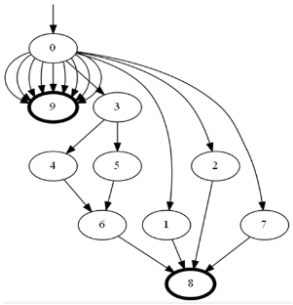
14.	Struktur *nested if* pada kasus IF then-else if EndIF. Bentuk program:
	`if ( kondisi ) {`
	`  statement kode program`
	`}else if ( kondisi ) {`
	`  statement kode program`
	`}else if ( kondisi ) {`
	`  statement kode program`
	`}`
		X	
			Terdapat aliran *node graph* yang tidak terhubung dengan *node awal*, akibatnya pembentukan *independent path* tidak sesuai dengan *coverage criteria*.

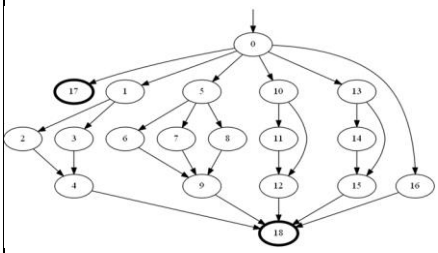
15.	Struktur *nested if* dalam bentuk IF then [IF then EndIF] - else [IF then EndIF] EndIF. Bentuk program:
	`if ( kondisi ) {`
	`  if ( kondisi ) {`
	`  statement kode program`
	`  }`
	`}else {`
	`  if ( kondisi ) {`
	`  statement kode program`
	`  }`
	`}`
		X	*Graph* tidak terbentuk dan muncul pesan:
			"<terminate> TRGenerate[
			Else without if"

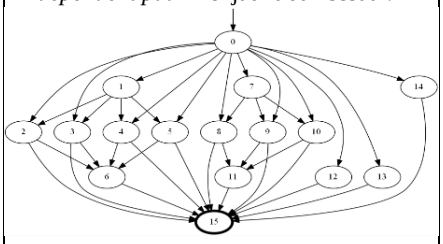
16.	Struktur *nested if* yang cukup rumit dengan tingkat kedalaman, bentuk program	X	*Graph* tidak sesuai, terdapat lebih dari satu *node* akhir percabangan. Analisa: semakin dalam if maka kesulitan untuk mengenali *node* yang merupakan bagian akhir dari pencabangan. Hal ini menyebabkan pembentukan *independent path* menjadi tidak sesuai
17.	Struktur *nested if* dalam bentuk IF then EndIF di dalam blok Switch Case. Bentuk program: `switch ( kondisi ) {`	V	-

No	Objektif Evaluasi terhadap Varian Logical Complexity Program	Hasil evaluasi	Keterangan

	`case nilai_1 : if ( kondisi ) { statement kode program } break`
	`case nilai_2 : if ( kondisi ) { statement kode program } break`
	`....`
	`default : statement kode program`
	`}`		

18.	Struktur Switch Case yang tidak semua *statement break* ada pada setiap kondisi *case*. Bentuk program:
	`switch ( kondisi ) {`
	`case nilai_1 :`
	`case nilai_2 :`
	`case nilai_3 :`
	`case nilai_4 :`
	`case nilai_5 :`
	`case nilai_6 :`
	`case nilai_7 : statement kode program, break`
	`case nilai_8 :`
	`case nilai_9 :`
	`case nilai_10 :`
	`case nilai_11 : statement kode program, break`
	`case nilai_12 : statement kode program, break`
	`default : statement kode program`
	`}`
		X	Bentuk *graph* dan isi *node* tidak sesuai dan memiliki dua *terminate node* sehingga pembentukan *independent path* menjadi tidak sesuai.
			 

19.	Struktur *nested if* dalam bentuk Switch Case yang di dalamya terdapat block If then else EndIF. Bentuk program:
	`switch ( kondisi ) {`
	`case nilai_1 : if ( kondisi ) { statement kode program }else { statement kode program } break`
	`case nilai_2 : if ( kondisi ) { statement kode program } else { statement kode program } break`
	`....`
	`default : statement kode program`
	`}`
		X	Bentuk CFG dan isi node tidak sesuai, karena graph tidak ada path yang membentuk percabangan dan CFG memiliki dua termanate node. Akibatnya pembuatan independent path menjadi tidak sesuai.
			 

20.	Struktur *nested if* dalam kasus Switch Case yang di dalamnya terdapat blok Switch Case. Bentuk program:
	`switch ( kondisi ) {`
	`case nilai_1 :`
	`switch ( kondisi ) {`
	`case nilai_1: statement kode program, break`
	`case nilai_2: statement kode program, break`
	`....`
	`default : statement kode program`
	`}`
	`case nilai_2 :`
	`switch ( kondisi ) {`
	`case nilai_1: statement kode program, break`
	`case nilai_2: statement kode program, break`
	`....`
	`default : statement kode program`
	`}`
	`....`
	`default : statement kode program`
	`}`
		X	Bentuk *graph* tidak sesuai karena semua *node* pada *block case* di dalam *case* masih tetap terhubung dengan *merge node*. Hal ini mengakibatkan pembuatan *independent path* menjadi tidak sesuai.
			 

Struktur Flow Repetation			
21.	Struktur For tanpa *nested loop*	V	-
22.	Struktur While-Do tanpa *nested loop*	V	-

No	Objektif Evaluasi terhadap Varian Logical Complexity Program	Hasil evaluasi	Keterangan
23.	Struktur Do-While tanpa *nested loop*	V	-
24.	Struktur *nested loop* dengan kasus For di dalam blok struktur For	V	-
25.	Struktur *nested loop* dengan kasus For di dalam blok struktur While-Do	V	-
26.	Struktur *nested loop* dengan kasus For di dalam blok struktur Do-While	V	-
27.	Struktur *nested loop* dengan kasus While-Do di dalam blok struktur For	V	-
28.	Struktur *nested loop* dengan kasus While-Do di dalam struktur While-Do	V	-
29.	Struktur *nested loop* dengan kasus While-Do struktur Do-While	V	-
30.	Struktur *nested loop* dengan kasus Do-While di dalam blok struktur For	V	-
31.	Struktur *nested loop* dengan kasus Do-While di dalam struktur While-Do	V	-
32.	Struktur *nested loop* dengan kasus Do-While di dalam struktur Do-While	V	-
Struktur Flow Kombinasi Looping dan Selection			
33.	Struktur If then EndIF di dalam blok struktur For	V	-
34.	Struktur If then EndIF di dalam blok struktur While-Do	V	-
35.	Struktur If then EndIF di dalam blok struktur struktur Do-While	V	-
36.	Struktur If then-else EndIF di dalam blok struktur For	V	-
37.	Struktur If then-else EndIF di dalam blok struktur While-Do	V	-
38.	Struktur If then EndIF di dalam blok struktur Do-While	V	-
39.	Struktur Switch Case di dalam blok struktur For	V	-
40.	Struktur Switch Case di dalam blok struktur While-Do	V	-
41.	Struktur Switch Case di dalam blok struktur Do-While	V	-

Berdasarkan hasil pengamatan pada 41 variasi struktur program, teridentifikasi 31 variasi struktur program (75%) yang mampu ditangani TRGeneration. Selain itu, hasil pengamatan menunjukkan bahwa jika pembentukan CFG tidak tepat, maka proses pembentukkan *independent path* menjadi tidak tepat. Sebaliknya jika CFG sudah tepat, yaitu menyimpan *statement code* pada *node* dan *edge*, maka pembentukkan *independent path* juga tepat. Tabel 2 menunjukkan ringkasan hasil pengamatan analisa kemampuan TRGeneration berdasarkan Tabel 1.

Tabel 2. Analisa Kemampuan TRGeneration

No.	Kategori Logical Complexity Program	Hasil Evaluasi	Jumlah
1	Struktur Sequence	Sesuai	4
2	Struktur Selection dengan case IF-Then-EndIF, IF-Then-Else-EndIF, IF-Then-Else IF-Else-EndIF dengan variable menggunakan tipe data int, float	Sesuai	5
3	Struktur selection dengan case IF-Then-Else IF-Then–Else IF-EndIF	tidak sesuai	2
4	Struktur selection dengan case IF-Then-Else [IF-Then-EndIF]-EndIF	tidak sesuai	2
5	Setruktur selection dengan menggunakan variable yang memiliki tipe data double.	tidak sesuai	1
6	Struktur Switch Case	tidak sesuai	4
7	Struktur selection dengan nested if yang rumit	tidak sesuai	1
8	Struktur Looping sederhana	Sesuai	4
9	Struktur nested loop, looping di dalam looping	Sesuai	9
10	Struktur kombinasi looping dan selection	Sesuai	9
		Total sesuai	31 (75%)
		Total tidak sesuai	10 (25%)
		Total struktur program	41

Pembahasan evaluasi TRGeneration pada setiap kategori *logical complexity* dari tabel 2 menggunakan teknik ilustrasi. Komponen ilustrasi evaluasi *tool* terdiri dari *source code* kasus uji dan informasi keluaran TRGeneration berupa CFG dan *independent path*.

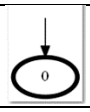
a. Struktur *sequence*.
   Pada struktur *sequence*, keluaran TRGeneration sudah sesuai dengan kaidah penulisan CFG dan identifikasi *independence path* sesuai dengan kriteria *statement coverage*. Pembahasan evaluasi tool pada struktur program ini dijelaskan pada ilutrasi 1.

Ilustrasi 1. Struktur *sequence*

Source Code

```
public   int   sisaTukarNominalPec1Kdgn50K10K5K(int
  nominalUang){
    int pec50K, pec10K, pec5K, pec1K;
    pec50K = nominalUang / 50000;
    pec10K = (nominalUang % 50000) / 10000;
    pec5K = ((nominalUang % 50000) % 10000) / 5000;
    pec1K = (((nominalUang % 50000) % 10000) % 5000)
       / 1000;
  return pec1K;
}
```

Output TRGerenation

CFG	Independent path
	[0]

Analisa: pada CFG, *node* 0 berisi code *sequence* mulai dari deklarasi *variable* sampai dengan *return*. *Path* yang teridentifikasi berjumlah 1, yaitu *node* 0 saja.

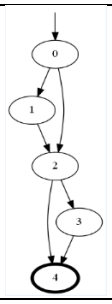
b. Struktur *selection* pada kategori nomor 2.
Pada struktur *selection* dengan kasus IF-Then-EndIF, IF-Then-Else-EndIF, dan IF-Then-Else IF-Else-EndIF memiliki keluaran TRGeneration yang sudah sesuai dengan kaidah penulisan CFG dan identifikasi *independence* path sesuai dengan kriteria *condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi TRGeneration pada struktur program ini dijelaskan pada ilutrasi 2 dan ilustrasi 3.

Ilustrasi 2. Struktur *selection* kasus IF-Then-EndIF

Source Code

```
public int cariMax3Bil_05(int A, int B, int C) {
  int max;
  max = A;
  if (max < B) {
    max = B;
  }
  if (max < C) {
    max = C;
  }
  return max;
}
```

Output TRGerenation

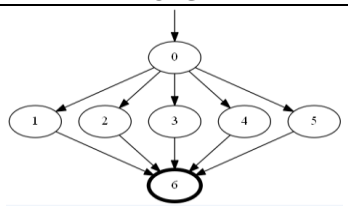
CFG	Independent path
	[0,2,4] [0,1,2,4] [0,2,3,4] [0,1,2,3,4]

Ilutrasi 3. Struktur *selection* kasus IF-Then-Else-EndIF dan IF-Then-Else IF-Else-EndIF

Source Code

```
public char hitNilaiMutu_06(double uts, double uas,
   double tugas, int hadir){
    char nilaiMutu = ' ';
    float nilai, nilaiHadir;
    nilaiHadir = hadir / 14 * 100f;
    nilai = (float) ((0.3 * uts) + (0.4 * uas) + (0.2
       * tugas) + (0.1 * nilaiHadir));
    if(nilai >= 85) {
       nilaiMutu = 'A';
    }else if (nilai >= 70) {
       nilaiMutu = 'B';
    } else if (nilai >= 55) {
       nilaiMutu = 'C';
    }else if (nilai >= 40) {
       nilaiMutu = 'D';
    } else {
       nilaiMutu = 'E';
    }
    return nilaiMutu;
}
```

Output TRGerenation

CFG	Independent path
	[0,1,6] [0,2,6] [0,3,6] [0,4,6] [0,5,6]

Analisa ketiga bentuk struktur program *loop* di atas: setiap pencabangan *node* pada CFG berisi *statement* program yang menunjukkan kondisi *true* atau *false*, dan aliran *edge* sesuai dengan alur program. Sedangkan *independent path* yang teridentifikasi sudah sesuai dengan alur *control flow program*. Hal ini berarti semua kemungkinan jalur/*path* eksekusi sudah semua teridentifikasi.

c. Struktur *selection* pada kategori 3.
Pada struktur *selection* dengan kasus if-then-Else IF-Then–Else IF-EndIF belum sesuai dengan kaidah penulisan CFG. Oleh karena CFG tidak tepat, maka identifikasi *independence path* tidak sesuai dengan kriteria *condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi tool pada struktur program ini dijelaskan pada ilutrasi 4.

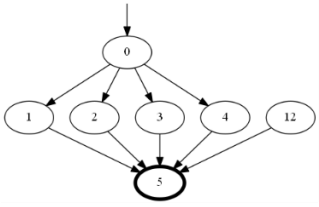
Ilustrasi 4. struktur *selection* dengan kasus If-Then-Else IF-Then–Else IF-EndIF

Source Code

```
public char hitNilaiMutu_06(double uts, double uas,
  double tugas, int hadir){
    char nilaiMutu = ' ';
    float nilai, nilaiHadir;
    nilaiHadir = hadir / 14 * 100f;
    nilai = (float) ((0.3 * uts) + (0.4 * uas) + (0.2
       * tugas) + (0.1 * nilaiHadir));
    nilaiMutu = 'E';
    if(nilai >= 85) {
       nilaiMutu = 'A';
    }else if (nilai >= 70) {
       nilaiMutu = 'B';
    } else if (nilai >= 55) {
       nilaiMutu = 'C';
    }else if (nilai >= 40) {
       nilaiMutu = 'D';
    }
    return nilaiMutu;
}
```

Output TRGerenation

CFG	Independent path

	[0,1,5] [0,2,5] [0,3,5] [0,4,5]

Analisa: pada CFG terdapat *node* yang tidak terhubung dengan *node* 0 sebagai alur pertama program, yaitu *node* 12. Hal ini terjadi karena proses penanganan TRGeneration pada struktur IF-ElseIF-ElseIF selalu memasangkan *statement* if dengan else. Oleh karena if terakhir tidak memiliki pasangan else maka tidak terjadi keterhubungan *graph* pada visuasi CFG. Akibatnya pada indentifikasi *independent path* tidak ada jalur dari 0 menuju 12. Padahal jika diamati, pada *source code* terdapat jalur menuju node 12 yang berisi *statement* "nilaiMutu = 'D';". Dengan demikian, pembuat TRGeneration tidak mampu menangani variasi ini.

d. Struktur *selection* pada kategori nomor 4.
Pada struktur *selection* dengan kasus IF-Then-Else [IF-Then-EndIF]-EndIF belum sesuai dengan kaidah penulisan CFG. Oleh karena CFG tidak tepat, maka identifikasi *independence path* tidak sesuai dengan kriteria *condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi TRGeneration pada struktur program ini dijelaskan pada ilutrasi 5.

Ilustrasi 5. Struktur *selection* kasus IF-Then-Else [IF-Then-EndIF]-EndIF

Source Code

```
public void tampilSuhu_10(int suhu){
  if (suhu <0) {
    System.out.println("Cair");
  }else {
    if ( suhu <= 100 ) {
        System.out.println("Padat");
    }else{
        System.out.println("Gas");
    }
  }
}
```

Output TRGerenation

Informasi	CFG	Independent path
<terminate> TRGenerate[Else without if	tidak terbentuk graph	Tidak teridentifikasi

Analisa: setiap eksekusi program TRGeneration menemukan *statement* else pertama kali, maka blok di dalam else akan dihapus semua data pada list *node*. Hal ini disebabkan karena pembuat TRGeneration menganggap bahwa else selalu berada di bagian akhir IF. Akibatnya untuk membaca IF yang kedua di dalam else tidak terdeteksi. Dengan demikian, pembuat TRGeneration tidak mampu menangani variasi ini.

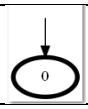
e. Struktur *selection* dengan menggunakan *variable* yang bertipe data *double* pada *statement* program di dalam blok if.
Pada struktur program ini, belum sesuai dengan kaidah penulisan CFG. Oleh karena CFG tidak tepat, maka identifikasi *independence path* tidak sesuai dengan kriteria *condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi TRGeneration pada struktur program ini dijelaskan pada ilutrasi 6.

Ilustrasi 6. Struktur *selection* kasus IF-Then-Else [IF-Then-EndIF]-EndIF

Source Code

```
public int cariMax3Bil_05(int A, int B, int C) {
  double max;
  max = A;
  if (max < B) {
    max = B;
  }
  return max;
}
```

Output TRGerenation

CFG	Independent path
	[0]

Analisa: *statement double* dianggap sebagai proses pembuatan CFG pada struktur *looping* dengan *keyword* "do". Hal ini disebabkan karena TRGeneration pada pembentukkan CFG berdasarkan *keyword* "do", "else", "case" dan "default". Apabila pengecekan sintaks pada *statement* program sama dengan salah satu *keyword* yang telah didefinisikan, maka *statement* program tidak akan dimasukan ke dalam *node*. Sehingga *statement* program dengan sintaks "double" yang berawalan "do" dianggap sebagai *keyword* "do". Akibatnya, *statement* program setelah *statement* "double" tidak mengalami proses yang normal untuk dimasukan ke dalam *node*. Dengan demikian, pembuat TRGeneration tidak mampu menangani variasi ini.

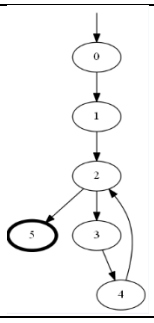
f. Struktur *looping* tanpa *nested loop*.
Pada struktur *looping* tanpa *nested loop*, keluaran TRGeneration sudah sesuai dengan kaidah penulisan CFG dan identifikasi *independence path* sesuai dengan kriteria *statement coverage, condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi tool pada struktur program ini dijelaskan pada ilutrasi 7, ilustrasi 8 dan ilustrasi 9.

Ilustrasi 7. Struktur *looping* kasus For

Source Code

```
Public void OutputBintang_21(int N){
  int i;
  for (i=1; i<=N; i++){
    System.out.print("*");
  }
}
```

Output TRGerenation

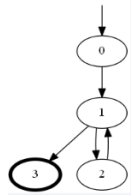
CFG	Independent path
	[0,1,2,5] [2,3,4,2] [3,4,2,3] [3,4,2,5] [4,2,3,4] [0,1,2,3,4]

Ilustrasi 8. Struktur *looping* dengan kasus While-Do

Source Code

```
public int jumBil_22(int bil[], int N){
  int i, sum;
  sum = 0;
  i=0;
  while (i<N){
    sum = sum + bil[i];
    i = i+1;
  }
  return sum;
}
```

Output TRGerenation

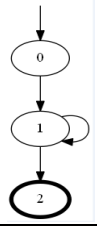
CFG	Independent path
	[0,1,2] [0,1,3] [1,2,1] [2,1,2] [2,1,3]

Ilustrasi 9. Struktur *looping* dengan kasus Do-While

Source Code

```
public int jumBil_23(int N){
  int sum, i;
  sum = 0;
  i = 0;
  do{
    sum = sum + 1;
    i = i + 1;
  }while (i<=N);
  return sum;
}
```

Output TRGerenation

CFG	Independent path
	[1,1] [0,1,2]

Analisa pada ketiga bentuk struktur program di atas: setiap pencabangan *node* berisi *statement* program yang menunjukkan kondisi *true* atau *false*, dan *edge* sesuai dengan alur program. *Statement* program pada akhir blok pengulangan divisualisasikan dengan *node* yang terhubung pada *node* pencabangan yang berisi kondisi pengulangan. *Independent path* yang teridentifikasi hanya pada jalur utamanya saja yang sudah sesuai dengan alur *control flow program* dan semua kemungkinan jalur eksekusi sudah semua teridentifikasi. Dengan demikian, identifikasi *independent path* pada tahap selanjutnya diperlukan proses untuk menghasilkan *path* yang utuh sebagai jalur eksekusi program. Namun berdasarkan kaidah teori pembuatan CFG dan *code coverage criteria* masih terpenuhi.

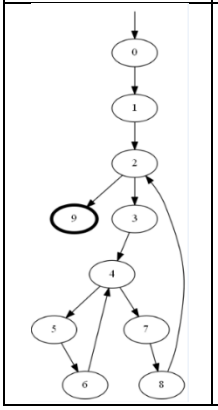
g. Struktur *looping* dengan nested loop.
Pada struktur *looping* dengan *nested loop*, keluaran TRGeneration sudah sesuai dengan kaidah penulisan CFG dan identifikasi *independence* path sesuai dengan kriteria *statement coverage, condition coverage*, *branch coverage* dan *path coverage*. Pembahasan evaluasi tool pada struktur program ini dijelaskan pada ilutrasi 10.

Ilustrasi 10. Struktur *looping* dengan *nested loop* pada kasus For

Source Code

```
public void OutputBintangSegiempat_33(int N){
  int i, j;
  for (i=1; i<=N; i++){
    for (j=1; j<=i; j++) {
      System.out.print("*");
    }
    System.out.println();
  }
}
```

Output TRGerenation

CFG	Independent path
	[0,1,2,5] [2,3,4,2] [3,4,2,3] [3,4,2,5] [4,2,3,4] [0,1,2,3,4]

Hasil analisa pada struktur *nested loop* menunjukkan hasil yang sama dengan analisa struktur *loop* tanpa *nested loop*. Hasil analisa tersebut adalah (1) setiap *node* pencabangan berisi kondisi yang menyatakan *true* atau *false*; (2) setiap *statement* program pada akhir pengulangan disimpan pada *node* yang memiliki *edge* yang terhubung dengan *node* pencabangan (kondisi pengulangan); (3) visualisasi CFG dan *independent path* pada *loop* yang berada di dalam blok *loop* memiliki pola yang sama dengan *loop* yang berada di luar.

Berdasarkan hasil analisa dan observasi kemampuan TRGeneration pada aktifitas di atas, TRGeneration hanya mampu menangani variasi program dengan struktur *sequence*, *selection* dengan struktur if-then-endif, if-then-else-endif, if-then-elseif-else-endif dan struktur *looping* dengan kasus tanpa *nested loop* dan *nested loop*. Sedangkan *selection* dengan struktur if-then-else yang menggunakan *nested if* dengan tingkat ke dalam di atas 3 level dan struktur *switch-case* tidak dapat ditangani oleh tool.

6. KESIMPULAN

Berdasarkan eksperimen dan pengamatan yang dilakukan dari 41 variasi *logical complexity*, TRGenerarion mampu menangani 31 variasi atau 75% dari kasus uji. Artinya, terdapat beberapa *logical complexity* yang tidak bisa ditangani, sehingga pada kondisi ini TRGeneration tidak direkomendasikan pada pengujian unit. Di sisi lain, kelebihan TRGeneration dalam hal visualisasi CFG sangat penting untuk merepresentasikan kelengkapan *test case*. Visualisasi CFG dipandang penting bagi *tester* untuk

mengetahui kelengkapan *test case* dalam satu siklus pengujian unit.

Jika TRGeneration akan digunakan untuk memeriksa kelengkapan *test case* dalam proses pengujian, ada 2 cara yang dapat dilakukan, yaitu:

1. Memodifikasi/ pengembangan TRGeneration agar 10 variasi *logical complexity* yang belum tertangani dapat dipenuhi. Selanjutnya TRGeneration dilengkapi dengan *sintax analyzer* agar dapat menganalis *code coverage*
2. Menggabungkan TRGeneration dengan *tool* lain yang memiliki kemampuan analisis *code coverage* yang sudah terbukti lengkap. Sehingga kemampuan visualisasi TRGeneration dapat dimanfaatkan untuk menganalisis *code coverage* melalui visualisasi CFG.

DAFTAR PUSTAKA

[1] Sommervile, Ian. 2016. "Software Engineering 10th". Pearson Education Limited. England

[2] S. Pressman, Roger. 2015. "Software Engineering: A Practitionr's Appriach 8th". New York. Amerika.

[3] https://github.com/evplatt/TRGeneration, Tanggal akses 21 Juni 2019.

[4] P. Hunt, Karl. 1979. "An Introduction to Structured Programming". Journal Behavior Research Methods& Instrumentation. Vol. 11(2), 229-233.

[5] Munir, Rinaldi. Lidya, Leony. 2016. "Algoritma dan Pemrograman dalam Bahasa Pascal, C dan C++ Ed. 6". Informatika

[6] Hendradjaya, Bayu. 2017. "Konsep Dasar Pengujian Perangkat Lunak". ITB Press. Bandung. Indonesia

[7] S. Arapidis, Charalampos. 2012 "Sonar Code Quality Testing Essential". Packt Publishing, Birmingham. Mumbai.

[8] Shelke, Sneha. Nagpure, Sangeeta. 2014. "The Study of Various Code Coverage Tools". International Journal of Computer Trends and Technology (IJCCT). Vol. 13. No. 1.