Deriving Test Cases

Metode ujicoba berbasis alur dapat diaplikasikan pada detail desain prosedural atau kode sumber. Ujicoba berbasis alur direpresentasikan menjadi beberapa tahapan :

1. Menggunakan desain atau kode sebagai pondasi, menggambarkan graf alur yang berhubungan.
   Dengan menggunakan contoh Gambar (a), diagram alur dapat dibentuk dengan memberikan nomor untuk PDL. Perintah-perintah yang ada akan mengacu pada *node* dari suatu diagram alur. Perintah yang telah diberi nomor terlihat pada gambar (b).

```
PROCEDURE average;

  *  This procedure computes the average of 100 or fewer
     numbers that lie between bounding values; it also computes the
     sum and the total number valid.

  INTERFACE RETURNS average, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;

  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
  i = 1;
  total.input = total.valid = 0;
  sum = 0;
  DO WHILE value[ i ] < > –999 and total.input < 100
     increment total.input by 1;
     IF value[ i ] > = minimum AND value[ i ] < = maximum
           THEN increment total.valid by 1;
                sum = sum + value[ i ]
           ELSE skip
     ENDIF
     increment i by 1;
  ENDDO
  IF total.valid > 0
     THEN average = sum / total.valid;
     ELSE average = –999;
  ENDIF
END average
```

Gambar (a) PDL for test design

2. Menetapkan komplesitas cyclomatic dari graf alur yang dihasilkan. Berdasarkan flow graph yang didapat pada Gambar (c), maka :
   V(G) = 6 region
   V(G) = 18 edges – 14 Nodes + 2 = 6
   V(G) = 5 Predicate nodes + 1 = 6

3. Menentukan himpunan basis dari alur independen yang linear.
   Nilai dari V(G) menyediakan sejumlah alur independen yang linear melalui struktur kontrol. Dalam kasus prosedur *Average* terdapat 6 :
   Path 1 : 1 – 2 – 10 – 11 – 13
   Path 2 : 1 – 2 – 10 – 12 – 13
   Path 3 : 1 – 2 – 3 – 10 – 11 – 13
   Path 4 : 1 – 2 – 3 – 4 – 5 – 8 – 9 – 2 – ...
   Path 5 : 1 – 2 – 3 – 4 – 5 – 6 – 8 – 9 – 2 – ...
   Path 4 : 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 2 – ...

   Tanda (...) mengindikasikan perulangan yang diperbolehkan. Dalam kasus ini node 2, 3, 5, 6, 10 merupakan *Predicates node*

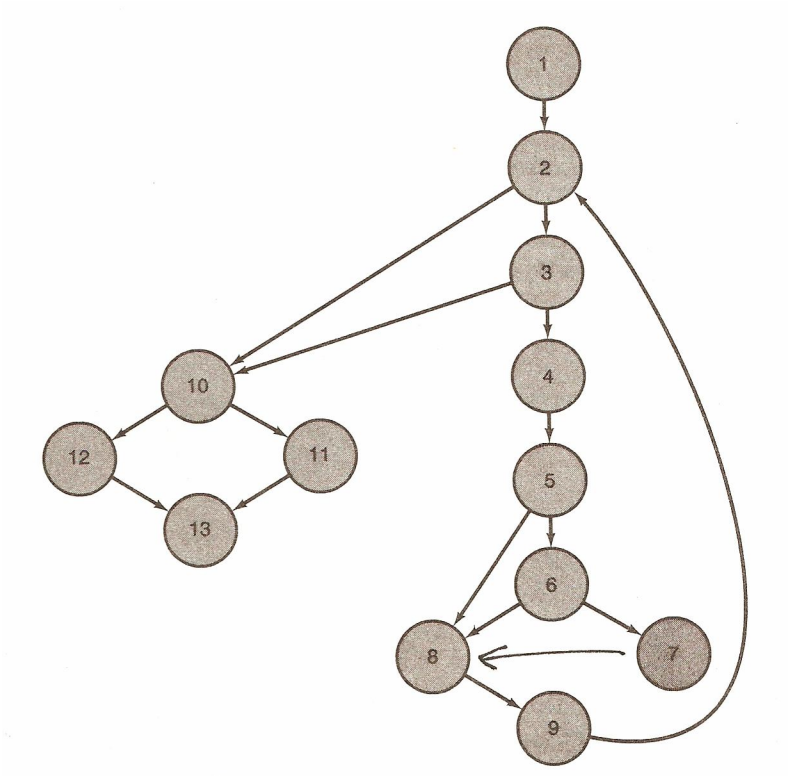
```
PROCEDURE average;

    * This procedure computes the average of 100 or fewer
      numbers that lie between bounding values; it also computes the
      sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;
    sum = 0;
2   DO WHILE value[ i ] < > –999 and total.input < 100 3
4     increment total.input by 1;
      IF value[ i ] > = minimum AND value[ i ] < = maximum 6
5       THEN increment total.valid by 1;
7               sum = sum + value[ i ]
        ELSE skip
8     ENDIF
      increment i by 1;
9   ENDDO
    IF total.valid > 0 10
11    THEN average = sum / total.valid;
12    ELSE average = –999;
13  ENDIF
END average
```

Gambar (b) Identifying Nodes



Gambar (c) *Flow Graph* of procedure Average

4. Menyiapkan kasus uji yang akan mengeksekusi setiap alur dalam himpunan basis Data harus dipilih sehingga kondisi pada predicate nodes dapat di uji dengan tepat

UJI KASUS Path 1:
nilai (k) = Input yang Valid, dimana k < i yang didefinisikan
nilai (i) = -999 dimana $2 \leq i \leq 100$
Hasil yang diharapkan :
Nilai rata-rata yang benar berdasarkan nilai k dan total yang tepat
Catatan :
Tidak dapat diujicoba secara terpisah, harus diujikan sebagai bagian dari uji Path 4, 5, dan 6

UJI KASUS Path 2:
nilai (1) = -999
Hasil yang diharapkan :
Nilai rata-rata/average = -999; total lainnya berisi nilai awal

UJI KASUS Path 3:
Berusaha untuk memproses nilai 101 atau lebih
100 nilai pertama harus valid
Hasil yang diharapkan :
Sama dengan UJI KASUS Path 1

UJI KASUS Path 4:
nilai (i) = Input yang Valid, dimana i < 100
nilai (k) < minimum dimana k < i
Hasil yang diharapkan :
Nilai rata-rata yang benar berdasarkan nilai k dan total yang tepat

UJI KASUS Path 5:
nilai (i) = Input yang Valid, dimana i < 100
nilai (k) > maksimum dimana $k \leq i$
Hasil yang diharapkan :
Nilai rata-rata yang benar berdasarkan nilai n dan total yang tepat

UJI KASUS Path 6:
nilai (i) = Input yang Valid, dimana i < 100
Hasil yang diharapkan :
Nilai rata-rata yang benar berdasarkan nilai n dan total yang tepat

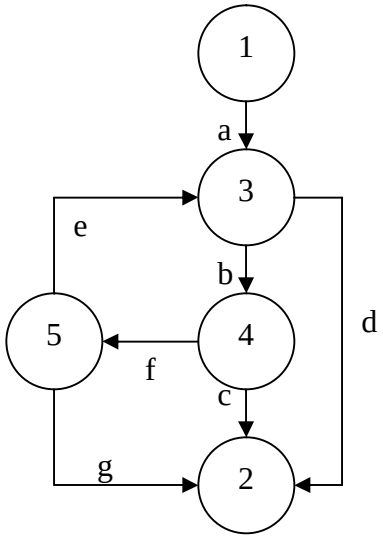
Setiap uji kasus dijalankan dan hasil yang diperoleh dibandingkan dengan hasil yang diharapkan. Ketika seluruh uji kasus telah dijalankan, maka penguji dapat memastikan bahwa setiap perintah yang ada telah dieksekusi sedikitnya 1 kali.

Graph Matrix

Prosedur untuk menghasilkan graf alur dan menentukan himpunan alur basis dapat di mekanisasi. Untuk membangun tool software yang membantu dalam ujicoba berbasis alur, struktur data yang disebut *graph matrix*. *Graph matrix* merupakan matrik persegi yang jumlah baris dan kolomnya sesuai dengan jumlah node pada graf alur. Setiap baris dan kolom mengacu kepada sebuah node dan isi dari matrix mengacu kepada edge, contoh gambar (d).

Graph Matrix merupakan representasi tabular daro flow graph, dengan menambahkan *link weight* untuk setiap input pada matrik, maka matrik graf menjadi alat bantu yang lebih berguna untuk mengevaluasi struktur kontrol program selama ujicoba. *link weight* menyediakan informasi tambahan mengenai aliran kontrol. Dengan bentuk sederhana, yaitu berikan nilai 1 jika ada koneksi antar node dan nilai 0 jika tidak ada koneksi. Hal lain yang dapat diketahui melalui matrik graf :

1. Kemungkinan suatu link/ edge akan dieksekusi
2. Waktu proses yang dibutuhkan selama traversal suatu link
3. Memori yang diperlukan selama traversal suatu link
4. Sumber daya yang diperlukan selama traversal suatu link

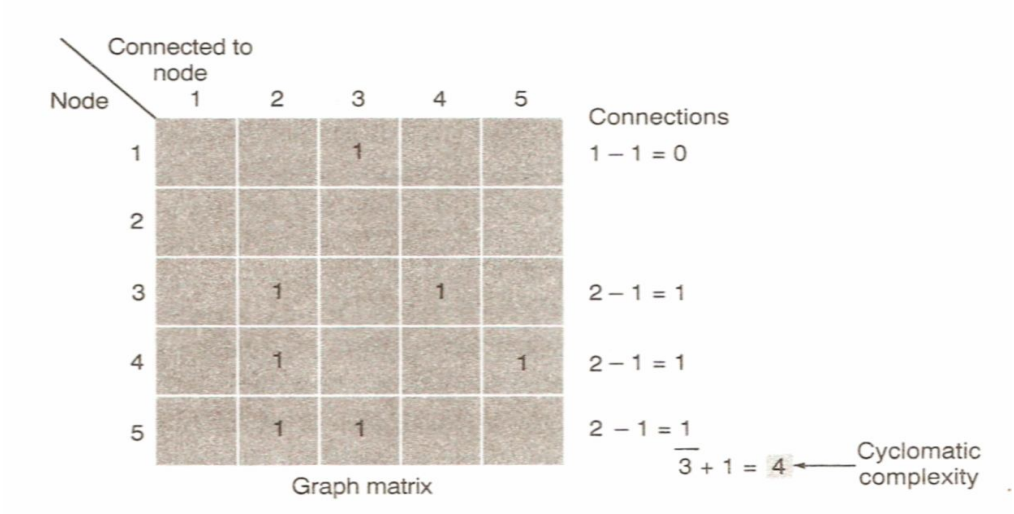


connected to node

node	1	2	3	4	5
1			**a**		
2					
3	**d**		**b**		
4	**c**				**f**
5	**g**	**e**			

Gambar (d) Flow graph dan graph matric-nya

Perhatikan Gambar (e) dibawah ini, setiap huruf telah diganti dengan angka 1, yang mengindikasikan terdapat koneksi. Setiap baris dengan 2 atau lebih input merepresentasikan *predicate node.* Lihat bagian kana tabel menunjukan *cyclomatic complexity*.



Gambar (d) *Connection Matrix*

UJICOBA BERBASIS STRUKTUR KONTROL (CONTROL STRUCTURE TESTING)

Ujicoba berbasis struktur kontrol merupakan variasi lain yang memperluas cakupan uji dan meningkatkan kualitas *whitebox testing*.

Ujicoba Kondisi (*Condition Testing*)

Ujicoba kondisi merupakan rancangan metode ujicoba kasus yang menguji kondisi logikal yang ada didalam modul program. Kondisi sederhana (*simple condition*) adalah variabel boolean atau ekspresi relational, yang mungkin dimulai dengan satu operator NOT (¬). Ekspresi relasional mempunyai bentuk :

E1 <operator relasional> E2

Dimana, E1 dan E2 merupakan ekspresi aritmatika dan <operator relational> adalah salah satu dari operator $<$, $>$, $=$, $\leq$, $\geq$, $\neq$. Kondisi gabungan (*compound condition*) merupakan gabungan dari dua atau lebih kondisi sederhana, operator boolean dan parantesis (tanda kurung). Diasumsikan operator boolean yang diperbolehkan

dalam kondisi gabungan termasuk OR (|), AND (&), dan NOT (¬). Kondisi tanpa ekspresi relasional direferensikan sebagai ekspresi boolean.

Karena itu jenis-jenis komponen yang mungkin dalam kondisi termasuk operator boolean, variabel boolean, sapasang parantesis boolean, operator relasional atau ekspresi aritmatika. Jika kondisi salah, maka sedikitnya satu komponen dari kondisi ada yang salah. Jenis-jenis kesalahan dalam kondisi meliputi :

- Kesalahan operator boolean (keberadaan operator boolean yang salah/berlebih/kurang)
- Kesalahan variable Boolean
- Kesalahan parantesis Boolean
- Kesalahan operator relasional
- Kesalahan ekspresi aritmatika

Metode ujicoba kondisi (*condition testing*) memfokuskan pada ujicoba setiap kondisi dalam program. Strategi ujicoba kondisi secara umum mempunyai 2 keuntungan, yaitu :

1. pengukuran dari ujicoba mencakup kondisi senderhana
2. Ujicoba mencakup kondisi dalam program menyediakan panduan untuk pembentukan ujicoba tambahan bagi program

Kegunaan dari ujicoba kondisi adalah untuk mendeteksi, tidak hanya kesalahan kondisi dalam suatu program, tetapi juga kesalahan lain dalam program. Jika suatu ujicoba ditentukan untuk suatu program *P*, maka diharapkan ujicoba ini juga efektif untuk mendeteksi kesalahan lain dalam *P*. Jika suatu ujicoba efektif dalam mendeteksi kesalahan dalam kondisi, maka akan efektif juga untuk mendeteksi kesalahan lain. Terdapat sejumlah strategi ujicoba kondisi. Ujicoba percabangan (*branch testing*) merupakan strategi ujicoba kondisi yang paling sederhana. Jika terdapat kondisi gabungan C, *true* dan *false* merupakan percabangan dari C, maka setiap cabang sederhana harus dieksekusi sedikitnya satu kali.

Ujicoba domain (*domain testing*) memerlukan 3 atau 4 kali ujicoba yang dilakukan untuk sebuah ekspresi relasional, misalkan saja ekspresi relasional dengan bentuk :

E1 <operator relasional> E2

Memerlukan 3 ujicoba untuk membuat nilai E1 lebih besar dari, sama dengan atau lebih kecil dari pada E2. Jika <operator relasional> salah dan E1 juga E2 benar, maka ketiga ujicoba ini menjamin pendeteksian dari kesalahan operator relasional. Untuk mendeteksi kesalahan dalam E1 dan E2, ujicoba yang menyebabkan nilai E1 lebih besar atau kurang dari E2 harus dibuat selisih sekecil mungkin.

Untuk ekspresi boolean dengan *n* variabel, maka diperlukan 2^n ujicoba (n > 0). Strategi ini dapat mendeteksi kesalahan operator boolean, variabel dan parantesis. Jika ujicoba dilakukan pada ekspresi boolean singular (yaitu setiap variabel boolean hanya muncul sekali) maka dibutuhkan ujicoba yang lebih sedikit dari 2^n ujicoba.

Branch and relational operator testing (BRO) merupakan teknik yang menjamin pendeteksian dari kesalahan percabangan dan operator relasional dalam kondisi yang disediakan, dimana seluruh variabel boolean dan operator relasional dalam kondisi muncul hanyan sekali. Strategi BRO menggunakan batasan kondisi. Untuk kondisi C, maka batasan kondisi untuk C dengan n kondisi sederhana didefinisikan sebagai (D1, D2, .., Dn), dimana Di ($0 < i \leq n$) merupakan simbol yang menspesifikasikan batasan dari hasil kondisi sederhana ke-i dalam kondisi C. Batasan kondisi D dalam kondisi C harus tercakupi dengan pengeksekusian C, jika selama eksekusi C, output dari setiap kondisi sederhana dalam C memenuhi batasan D yang ditetapkan. Misal diketahui contoh kondisi berikut (contoh 1):

C1 : B1 & B2

Dimana :
B1 dan B2 : variabel boolean.
Batasan kondisi untuk C1 adalah (D1, D2),
dimana untuk setiap D1 dan D2 : *true* (t) atau *false* (f).
Nilai (t, f) merupakan batasan kondisi untuk C1 dan tercakupi oleh ujicoba yang menyebabkan nilai dari B1 menjadi *true* dan nilai B2 menjadi *false*.

Strategi ujicoba BRO membutuhkan himpunan batasan {(t, t), (f, t), (t, f)} tercakupi dengan pengeksekusian C1. Jika C1 salah disebabkan karena satu atau lebih kesalahan operator boolean, sedikitnya satu dari pasangan batasan akan menyebabkan C1 gagal.

Perhatikan contoh berikut (contoh 2):

C2 : B1 & (E3 = E4)

Dimana :
B1 : ekspresi boolean
E3, E4 : ekspresi aritmatika.
Batasan kondisi untuk C2 adalah (D1, D2)
dimana : setiap D1 adalah t atau f, dan D2 adalah >, =, atau <,

maka dapat dibentuk himpunan batasan untuk C2 dengan memodifikasi batasan yang didefinisikan untuk C1. Dimana t untuk (E3=E4) mengamplikasikan =, dan f untuk (E3=E4)mengaplikasikan < atau >. Dengan mengganti (t, t) dan (f, t) dengan (t, =) dan (f, =) dan mengganti (t, f) dengan (t, <) dan (t, >) maka himpunan batasan untuk C2 adalah {(t, =), (f, =), (t, <), (t, >)}, maka himpunan batasan tadi akan menjamin pendeteksian kesalahan operator boolean dan operasional dalam C2.

Perhatikan contoh berikut (contoh 3) :

C3 : (E1 > E2) & (E3 = E4)

Dimana :
E1, E2, E3, E4 : operator aritmatika.
Batasan kondisi untuk C3 dalam bentuk (D1, D2)
dimana setiap D1 dan D2 adalah >, =, <.

Maka dapat dibentuk himpunan batasan sebagai berikut : {(>, =), (=, =), (<, =), (>, >), (<, <)}

Ujicoba Aliran Data (*Data Flow Testing*)

Metode ujicoba aliran data memilih alur dari program berdasarkan lokasi pendefinisian dan penggunaan variabel dalam program. Untuk mengilustrasikan pendekatan ujicoba aliran data, asumsikan bahwa setiap perintah dalam program ditentukan sejumlah perintah yang unique dan setiap fungsi tidak merubah parameternya atau variabel global. Untuk perintah dengan S sebagai nomor perintah :

DEF (S) = {X| perintah S mengandung definisi X}
USE (S) = {X| Perintah S mengandung penggunaan X}

Jika perintah S merupakan perintah *if* atau *loop*, himpunan DEF = kosong (empty) dan himpunan USE berdasarkan pada kondisi perintah S. Definisi variable X pada perintah S akan ada pada perintah S` jika ada alur(path) dari perintah S ke perintah S` yang tidak mengandung pendefinisian lain dari variable X.

Definition Use Chain (DU chain) dari variable X mempunyai bentuk [X, S, S`] dimana S dan S` adalah nomor perintah dan X dalam DEF(S) dan USE(S`) dan definisi X dalam perintah S ada dalam perintah S`. Contoh sederhana dari strategi ujicoba aliran data adalah untuk memastikan setiap DU chain tercakupi sedikitnya satu kali. Ujicoba DU tidak menjamin cakupan dari seluruh bagian program. Misalkan perintah *if-then-else* dan tidak menyertakan perintah untuk *else* maka ujicoba DU tidak diperlukan untuk kondisi *else* ini.

Strategi ujicoba aliran data berguna untuk menyeleksi ujicoba alur (path) program yang mengandung *nested if* dan perintah perulangan. Perhatikan contoh program berikut :

```
proc X
    B1;
    do while C1
        if c2  then
            if C4 then B4;
                else B5;
            endif;
        else
            if C3 then B2;
                else B3;
            endif;
        endif;
    enddo;
    B6;
End proc;
```

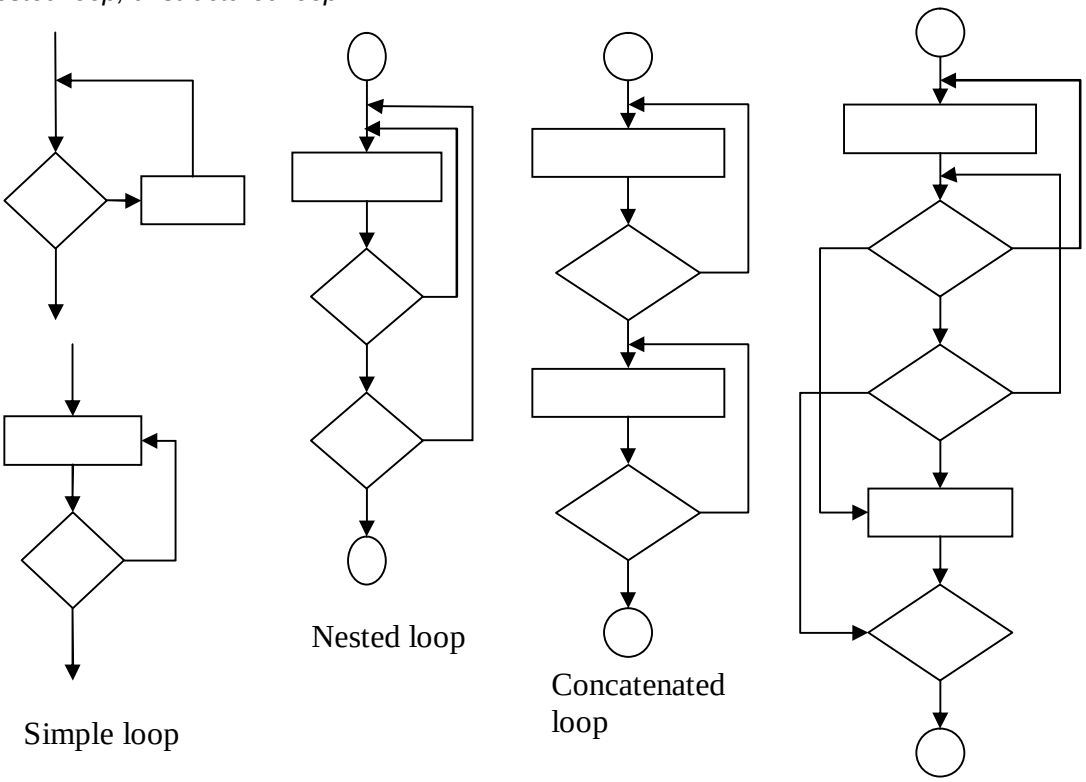
Untuk mengaplikasikan strategi DU untuk memilih ujicoba alur dari diagram kontrol, perlu mengetahui definisi dan penggunaan variabel dalam setiap kondisi atau blok dalam program. Diasumsikan bahwa variabel X didefinisikan pada perintah terakhir dari blok B1, B2, B3, B4, B5 dan digunakan dalam perintah pertama dari blok B2, B3, B4, B5, B6. Strategi ujicoba DU membutuhkan eksekusi dari alur terpendek dari setiap Bi, $0 < i \leq 5$, untuk setiap Bj, $2 < j \leq 6$ (Ujicoba jenis ini juga mencakup setiap penggunaan variabel X dalam kondisi C1, C2, C3, C4). Walaupun terdapat 25 rantai Duvariabel X, hanya dibutuhkan 5 alur untuk mencakup rantai DU ini. Alasannya adalah 5 alur ini dibutuhkan untuk mencakup rantai DU X dari Bi, $0 < i \leq 5$, ke B6 dan rantai DU lainnya dapat dicakup dengan membuat alur ini berulang.

Jika mengaplikasikan strategi percabangan untuk menyeleksi ujicoba alur program diatas, dengan ujicoba BRO perlu diketahui struktur dari setiap kondisi atau blok. (Setelah memilih alur program, perlu menentukan apakah alur tersebut mungkin untuk program misal : terdapat sedikitnya satu input yang dilakukan untuk setiap alur)

Jika perintah dalam program dihubungkan antara satu dengan lainnya berdasarkan definisi dan penggunaan variabel, maka pendekatan ujicoba aliran data efektif untuk penjagaan kesalahan. Bagaimanapun, masalah pengukuran kecakupan (*coverage*) dan pemilihan alur (path) uji untuk ujicoba aliran data lebih sulit dibandingkan dengan ujicoba kondisi.

Ujicoba Perulangan (*Loop Testing*)

Ujicoba perulangan (*Loop Testing*) merupakan ujicoba *whitebox* yang memfokuskan secara khusus pada validitas pembentukan loop. Terdapat 4 jenis loop yang berbeda, yaitu : *simple loop, concatenated loop, nested loop, unstructured loop.*



Simple loop

Nested loop

Concatenated loop

Unstructured loop

Simple loop, serangkaian ujicoba harus diaplikasikan pada loop sederhana, dimana n merupakan jumlah maksimum yang diperbolehkan dalam loop :

1. Lewati loop secara keseluruhan (*skip the loop entirely*)
2. Hanya satu yang melalui loop (*only one pass through the loop*)
3. 2 melalui loop (*two passes through the loop*)
4. m melalui loop, jika m < n (*m passes the loop where m < n*)
5. n – 1, n, n+1 passes the loop

Nested loop, bila menerapkan ujicoba untuk loop sederhana pada nested loop, maka jumlah dari ujicoba yang mungkin berkembang secara geometri sejalan dengan peningkatan level dari nested. ***Beizer*** menyarankan pendekatan yang akan mengurangi jumlah ujicoba :

1. Mulai dari loop terdalam, Tetapkan loop lainnya pada nilai minimum
2. Lakukan uji loop sederhana untuk loop terdalam dan loop luar tetap pada parameter iterasi minimum. Tambahkan ujicoba lainnya untuk *out-of-range* atau nilai lainnya (*exclude value*)
3. Lakukan mengarah keluar, dan tetap mempertahankan loop luarnya dalam nilai minimum dan nested loop lainnya dengan nilai yang sesuai.
4. Lanjutkan sampai seluruh loop teruji.

Concatenated loop, dapat diuji dengan menggunakan pendekatan yang didefinisikan untuk loop sederhana, jika setiap loop saling independen. Jika loop digabungkan (*concatenated*) dan counter untuk loop1 digunakan sebagai nilai awal dari loop2, maka loop saling bergantung dan pendekatan untuk nested loop direkomendasikan

Unstructured loop, Jika mungkin sebaiknya loop jenis ini didesain ulang untuk merefleksikan kegunaan pembentukan struktur program.