PERANCANGAN DAN PEMBUATAN *CASE TOOL SOFTWARE TESTING* MENGGUNAKAN METODE *STATIC DATA FLOW ANALYSIS* (STUDI KASUS POGRAM BERBASIS C++)

Yudhi Purwananto, Arif Bramantoro, Luluk Harini
Jurusan Teknik Informatika,
Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember
Kampus ITS, Jl. Raya ITS, Sukolilo – Surabaya 60111, Telp. + 62 31 5939214, Fax. + 62 31 5913804
Email : yudhi@its-sby.edu

ABSTRAK

Penentuan kualitas atau mutu suatu perangkat lunak mutlak diperlukan. Kualitas suatu perangkat lunak dapat dinyatakan baik bila telah sesuai dengan requirement pengguna tanpa mengabaikan segi strukturalnya. Dengan tidak mengabaikan fungsional program, uji coba struktural juga harus dilakukan karena uji coba fungsional tak mampu menangani hal-hal yang berhubungan dengan struktural program.. Uji coba fungsional tidak dapat menentukan apakah suatu bagian program telah dieksekusi atau tidak. Dengan demikian, program yang tidak dieksekusi selama pengujian akan tersembunyi dalam package program dan bila didalamnya terdapat kesalahan maka kesalahan tersebut akan tersembunyi dalam jangka waktu tertentu. Data flow analysis adalah suatu metode yang digunakan untuk mengumpulkan informasi dalam program tanpa mengeksekusi program yang diuji coba. Metode ini merupakan bagian dari metode pengujian secara structural yang sangat efektif untuk menemukan kesalahan yang berupa data flow anomaly dengan cara memeriksa kode program. Dengan metode ini, seluruh pernyataan program yang diuji coba akan dianalisa.

Metodologi yang digunakan dalam penelitian ini terdiri dari beberapa tahapan. Yang dilakukan pertama kali adalah melakukan parsing terhadap file input untuk dijadikan file teks dengan format tertentu yang sudah terpisah menjadi token yang dapat berdiri sendiri. Selanjutnya akan diambil daftar nama fungsi dalam program untuk dijadikan input fungsi. Selanjutnya isi fungsi input akan direpresentasikan menjadi sebuah graph. Dari graph inilah dapat dicari keberadaan data flow anomaly dalam fungsi dan dibuat visualisasi aliran program dalam suatu control flow graph. Tahap terakhir yang dilakukan adalah uji coba dan evaluasi.

Dari uji coba dan evaluasi yang dilakukan pada berbagai macam jenis fungsi, baik fungsi sederhana yang hanya melibatkan pernyataan-pernyataan sekuensial, fungsi yang melibatkan percabangan, perulangan maupun fungsi yang melibatkan pemanggilan terhadap fungsi lain didapatkan data flow anomaly maupun control flow graph dengan benar.

Kata Kunci : *uji coba fungsional, uji coba struktural,data flow anomaly, control flow graph(CFG).*

1. PENDAHULUAN

Uji coba perangkat lunak sangat dibutuhkan dalam menentukan baik buruknya kualitas suatu perangkat lunak. Hal ini bisa dilakukan dengan dua cara yaitu ujicoba fungsional dan ujicoba struktural. Kedua jenis ini tidak dapat saling menggantikan karena keduanya mempunyai titik berat yang berbeda. Suatu perangkat lunak dikatakan bagus kualitasnya bila secara fungsional telah memenuhi requirement pengguna dan bagus dari segi strukturalnya.

Uji coba fungsional tak dapat menangani hal-hal yang berhubungan dengan struktural program. Untuk program dengan banyak kombinasi jenis input tidaklah tepat diuji dengan metode fungsional. Selain itu uji coba fungsional juga tak dapat menentukan apakah suatu bagian program telah dieksekusi atau tidak[3] . Dengan demikian, program yang tidak dieksekusi selama pengujian akan tersembunyi dalam package program dan bila didalamnya terdapat kesalahan maka kesalahan tersebut akan tersembunyi dalam jangka waktu tertentu.

Data flow analysis merupakan bagian dari uji coba struktural (*white-box testing*). Dengan menggunakan metode ini dalam pengujian, *def-use* dari variabel yang terlibat dalam program bisa diketahui sehingga *d-d, u-r* dan *d-u anomaly* dapat diketahui untuk kemudian dapat diperbaiki.

Penelitian ini disusun untuk dapat melakukan uji coba terhadap perangkat lunak yang menggunakan bahasa pemrograman C++. Dengan demikian dapat diketahui keberadaan *data flow anomaly* dalam

program dan dapat membantu dalam menilai kualitas suatu perangkat lunak.

2. PENGUJIAN PERANGKAT LUNAK

Sejumlah *rule*/aturan yang dapat berperan sebagai tujuan pengujian perangkat lunak meliputi :[4]

a. Pengujian merupakan suatu proses pengeksekusian program dengan tujuan untuk menemukan kesalahan/*error.*
b. *Test case* yang baik adalah yang mempunyai probabilitas yang tinggi untuk menemukan kesalahan/error yang belum ditemukan sebelumnya.
c. Pengujian yang berhasil adalah pengujian yang menemukan kesalahan/error yang belum ditemukan sebelumnya.

Jika pengujian dilakukan dengan sukses maka pengujian tersebut akan menemukan kesalahan/error dalam perangkat lunak. Keuntungan yang lain adalah pengujian memperlihatkan bahwa fungsi-fungsi dalam perangkat lunak bekerja sesuai dengan spesifikasi, tingkah laku program dan *performance*-nya sesuai dengan yang diinginkan. Tetapi, pengujian tidak dapat menunjukkan adanya *error* dan *defect,* hanya dapat menunjukkan bahwa *error* dan *defect* tersebut ada[4].

2.1. PRINSIP-PRINSIP PENGUJIAN

Berikut ini prinsip-prinsip pengujian perangkat lunak :[4]

1. Semua pengujian seharusnya dapat ditelusuri ke *requirement* konsumen. Sebagaimana telah diketahui, pengujian perangkat lunak bertujuan untuk menemukan kesalahan. Kesalahan yang paling berat adalah kesalahan yang menyebabkan program gagal untuk mencapai *requirement*.
2. Pengujian harus direncanakan jauh sebelum pengujian dilaksanakan. Rencana pengujian dapat dimulai segera setelah model *requirement* selesai. Dengan demikian, semua pengujian dapat direncanakan dan dirancang sebelum kode program dibuat.
3. Pengujian seharusnya dimulai dari ukuran kecil, baru kemudian ke ukuran yang lebih besar. Pengujian pertama yang direncanakan dan dilakukan secara umum terfokus pada komponen-komponen secara individu. Setelah itu dilakukan pengujian pada sekumpulan komponen yang sudah terintegrasi dan kemudian pada keseluruhan sistem.
4. Tidak perlu dilakukan pengujian yang mendalam.
5. Pengujian seharusnya dilakukan oleh pihak ketiga yang bersifat independen. Dengan demikian, pengujian menjadi lebih efektif dan kemungkinan ditemukan kesalahan leih besar daripada diuji sendiri oleh pembuat perangkat lunak tersebut.

2.2. CIRI-CIRI PENGUJIAN YANG BAIK

Untuk mendapatkan hasil yang baik maka pengujian yang dilakukan harus dilakukan secara optimal. Oleh sebab itu, pengujian tersebut harus memiliki ciri yang baik, yaitu[4]:

1. Memiliki kemampuan yang tinggi untuk menemukan kesalahan. Untuk mencapai tujuan ini, penguji harus mengerti perangkat lunak dan usaha untuk mengembangkan sebuah *mental picture* (bayangan) mengenai bagaimana perangkat lunak mungkin gagal.
2. Tidak berlebih-lebihan. Waktu dan sumber daya pengujian bersifat terbatas. Dengan demikian, setiap pengujian harus memiliki tujuan yang berbeda sehingga tidak terjadi pemborosan dari segi waktu maupun sumber daya pengujian.
3. Harus mempunyai sasaran yang tepat. Bila terdapat sekumpulan pengujian yang mempunyai tujuan yang hampir sama maka hanya sebagian dari pengujian tersebut yang dilakukan. Hal ini disebabkan oleh terbatasnya waktu dan sumber daya .
4. Tidak terlalu sederhana dan tidak terlalu kompleks. Meskipun kadang-kadang memungkinkan untuk menggabungkan sejumlah pengujian ke dalam satu *test case*, efek samping dari hal semacam ini adalah adanya kemungkinan tersembunyinya kesalahan.

Secara tradisional terdapat dua pendekatan pokok untuk menguji perangkat lunak yaitu *black-box testing* dan *white-box testing*[2][4].

2.3. BLACK-BOX TESTING

Black-box testing disebut juga *functional testing* atau *behavioral testing*, dimana perangkat lunak diuji melalui semua range input dan outputnya diamati kebenarannya[2][4]. Yang menjadi pusat perhatian adalah apakah output sudah sesuai dengan yang diharapkan dan telah memenuhi semua *requirement* pengguna. Jadi tidak dipermasalahkan bagaimana cara mendapatkan output. Dengan kata lain, meskipun jenis pengujian ini didesain untuk menemukan kesalahan/error, *black-box testing* digunakan untuk mendemonstrasikan bahwa fungsi-fungsi perangkat lunak bekerja, input dapat diterima dengan baik dan output sesuai yang diinginkan. *Black-box testing* memeriksa aspek dasar dari sebuah sistem dengan sedikit perhatian terhadap struktur logika internal dari perangkat lunak.

Diagram *Black-box* merupakan suatu contoh dari pengujian dengan menggunakan pendekatan *black-box*. Diagram tersebut menjelaskan bahwa dengan memasukkan input bilangan tertentu ke dalam suatu

fungsi yaitu *square root* akan dihasilkan output akar kuadrat dari input yang dimasukkan. Dengan demikian yang diuji hanyalah fungsional program yaitu apakah benar dengan input yang dimasukkan menghasilkan output sesuai dengan yang diharapkan.

Meskipun mempunyai beberapa keuntungan, *black box testing* mempunyai beberapa kelemahan. Kelemahan-kelemahan tersebut adalah:

a. Untuk sistem yang *real-life*, terlalu banyak jenis input yang berbeda. Hal ini menghasilkan kombinasi *test case* yang banyak dan tidak tepat untuk diuji dengan menggunakan metode *black-box testing.*
b. Tidak dapat menentukan apakah bagian dari *code* telah dieksekusi. Kode program yang tidak dieksekusi selama pengujian akan tersembunyi dalam *package* perangkat lunak.
c. Fakta empiris menunjukkan bahwa *black-box testing* tidak dapat menemukan sebanyak mungkin kesalahan/*error* seperti bila kombinasi metode pengujian yang dilakukan.

2.4. WHITE-BOX TESTING

Untuk mengatasi kelemahan-kelemahan pada *black-box testing* maka digunakan *white-box testing* sebagai tambahan pada *black-box testing*. *White-box testing* juga dikenal dengan sebutan *glass box, structural, clear box dan open box testing*[2][4]. Strategi *white-box testing* termasuk perancangan tes/pengujian seperti, setiap baris kode program dieksekusi setidaknya satu kali atau setiap fungsi harus diuji tersendiri. *White-box testing* menggunakan pengetahuan khusus/ tertentu mengenai kode pemrograman untuk memeriksa output. Pengujian dengan cara ini akan akurat bila penguji benar-benar mengerti apa yang harus dilakukan oleh program.

Sangat sedikit pengujian dengan metode *white-box testing* ini dapat dilakukan tanpa memodifikasi program, mengubah beberapa nilai untuk memaksa melakukan eksekusi pada *path* yang berbeda atau membangkitkan semua *range* input untuk menguji fungsi tertentu. Secara tradisional, modifikasi ini telah dibuat dengan menggunakan *debugger* yang interaktif atau dengan benar-benar mengubah kode program. Mungkin ini memadai untuk program berukuran kecil, namun tidak demikian untuk program yang lebih besar. *Debugger* tradisional sangat berpengaruh terhadap *timing* sehingga kadang-kadang aplikasi yang besar tidak dapat berjalan tanpa modifikasi yang besar.

Wh*ite box testing* mempunyai beberapa kriteria sebagai berikut :

1. *Statement Coverage*
   Setiap *statement*/pernyataan harus dieksekusi minimal satu kali.
2. *Edge Coverage*
   Setiap *edge* harus ditelusuri minimal satu kali
3. *Condition Coverage*
   Untuk operator logika biner (&&, ||), komponen-komponennya dievaluasi tersendiri dalam semua kombinasi nilai (benar atau salah)
4. *Relational Coverage*
   Untuk operator-operator relasional (<, >, <=, >=), kondisi sama dengan diperlakukan sebagai sebuah percabangan yang berbeda.
5. *Path Coverage*
   Setiap *path* yang mungkin dieksekusi minimal sekali.

Ada beberapa *tool* pengujian dengan metode *white-box testing* tanpa memodifikasi kode program dan tanpa mengeluarkan ongkos tambahan dari *debugger* interaktif. Keuntungan dari pendekatan ini adalah :

a. Tool ini mempercepat pengujian dan pencarian kesalahan/ *debugging* karena tidak perlu menunggu untuk menguji kode yang support untuk disisipkan ke dalam program. Dalam pengembangan perangkat lunak komersial, dengan beberapa developer dan suatu bagian yang bertanggung jawab terhadap pengujian dan integrasi, cara ini dapat menhemat waktu yang tentu saja sangat berarti.
b. Penggunaan terbaik dapat dibuat dari lingkungan test. Lebih cepat pengujian dilakukan maka lebih murah biaya yang dikeluarkan dalam proses pengujian.
c. Kode program mungkin tidak tersedia untuk semua perangkat lunak. Biasanya suatu organisasi, lembaga atau dunia industri memakai produk atau perangkat lunak yang dibuat oleh organisasi lain atau *software developer.* Biasanya tidak mengikutsertakan *source code* dari perangkat lunak tersebut.

3. PERANCANGAN DAN IMPLEMENTASI

Dalam pengembangan penelitian ini, dibangun beberapa tahap rancangan dan implementasinya yang dapat dijelaskan sebagai berikut.

3.1. DATA FLOW ANALYSIS (DFA)

Data flow Analysis merupakan suatu proses untuk mengumpulkan informasi pada saat *run time (run-time information)* mengenai data dalam program tanpa benar-benar mengeksekusinya [5][6].

Dalam *data flow analysis t*erdapat 3 macam status variabel yaitu *defined, referenced dan undefined*. Suatu variabel dikatakan *defined* bila variabel tersebut telah diberi nilai, *referenced* bila nilainya telah digunakan atau diacu oleh variabel lain, *undefined* bila nilai tak diketahui atau tak dapat dicapai.

Serangkaian *action* atau penggunaan variabel yang tidak jelas disebut *data flow anomaly*[5][3] . Hal ini mengindikasikan kemungkinan adanya

kesalahan pemrograman. Terdapat 3 macam *data flow anomaly* yaitu ur,du dan dd *anomaly.* Disebut ur *anomaly* jika suatu variabel dalam status *undefined* diacu oleh variabel lain atau digunakan dalam program, du *anomaly* jika suatu variabel telah didefinisikan tapi tak pernah diacu atau digunakan dalam program, dd *anomaly* jika suatu variabel didefinisikan lalu didefinisikan lagi sebelum digunakan atau diacu variabel lain dalam program.

Data flow analysis diimplementasikan dengan 2 cara yaitu *static Data flow analysis* dan *dynamic Data flow analysis*. Pada *static Data flow analysis*, analisa dilakukan tanpa mengeksekusi program dan lebih merupakan pendekatan pena dan kertas, keberadaan *data flow anomaly* diperiksa melalui kode program yang ada[7]. Sedangkan pada *dynamic data flow analysis*, analisa dilakukan dengan cara menginstrumentasikan program, yaitu menyisipkan beberapa pernyataan ke dalam program untuk mendapatkan informasi yang dibutuhkan.

Tak satupun dari kedua cara tersebut yang sempurna. Keduanya saling melengkapi. Berikut ini kelemahan masing-masing cara:

Metode statis :

- Metode statis sulit menangani tipe data array.
- Menampakkan semua *data flow anomaly.*

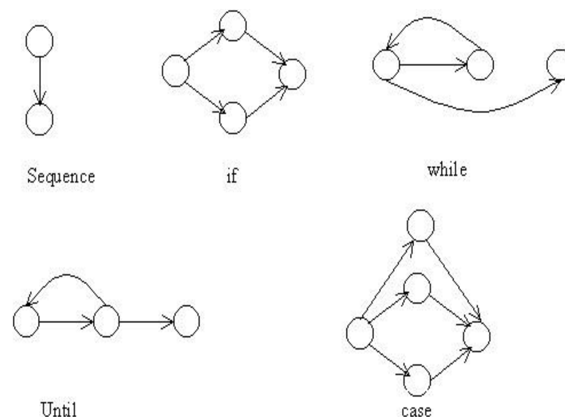
Metode dinamis:

- Metode dinamis bisa menangani array tanpa banyak masalah.
- Hanya mendeteksi *anomaly* sepanjang *path* yang benar-benar dieksekusi.
- Dapat menangani *object* dengan mudah.

3.2. CONTROL FLOW GRAPH (CFG)

Control flow graph (CFG) merupakan grafik yang merepresentasikan kontrol aliran dari sebuah program[5]. Grafik ini terdiri dari sekumpulan titik atau yang biasa disebut *node* dan sekumpulan garis berarah yang disebut dengan *edge*.

Node menunjukkan pernyataan (*statement)* program sedangkan *edge* (garis berarah) menunjukkan kontrol aliran dari pernyataan-pernyataan yang bersangkutan. Setiap *flow graph* menggambarkan aliran kontrol logis (*logical control flow)*, dengan notasi sebagaimana yang terdapat pada gambar 1. Untuk memberi informasi tambahan untuk *node* maupun *edge*, bisa digunakan label. Pada aplikasi yang dibuat ini, label akan digunakan untuk memberi informasi baris dari setiap pernyataan yang diwakili oleh satu node.



Gambar 1. Control Flow Graph (CFG)

3.3. PARSING DATA TESTING

Dalam pencarian *data flow anomaly* maupun pembangkitan *control flow graph (CFG)*, terdapat beberapa langkah yang harus dilakukan.

Langkah pertama adalah pemasukan data input. Data yang dimasukkan adalah berupa kode program berbahasa C++, dengan kata lain file tersebut berekstensi cpp atau CPP. Pengguna tidak dapat mengedit kode program. Kode program dianggap sudah valid, artinya file tersebut sudah melalui proses kompilasi dan bebas dari kesalahan atau *compilation error.* Segera setelah file input dimasukkan, proses parsing terhadap isi file dilakukan untuk menghasilkan suatu file teks dengan format tertentu, dimana antara token yang dapat berdiri sendiri telah terpisah dengan *operator-operator* yang ada. Misalnya token *return(0)*, token ini akan dipisahkan menjadi *return (0).* Pada langkah ini akan terjadi proses parsing yang tediri dari 5 proses yaitu:

- Pemisahan token dan operator.
  Pada langkah ini setiap token akan dipisahkan dari operator yang melekat padanya.
- Pembandingan token *keyword* dan token uji.
  Langkah ini dilakukan untuk memastikan apakah suatu token merupakan variabel atau tidak.
- Pembandingan karakter token uji dengan operator.
  Langkah ini merupakan bagian dari langkah pemisahan token dan operator. Jadi setiap karakter dalam token akan dibandingkan dengan sekumpulan operaor yang ada dan jika sama maka operator tersebut akan dipisahkan dari token yang bersangkutan. Rumusan yang dipakai adalah:
  $\mathbf{token=token_0 + operator_0 + token_1 + \ldots}$
- Pengabaian baris dan pernyataan.
  Token yang merupakan bagian dari baris yang diabaikan tidak akan diperiksa dan tidak akan dibandingkan dengan *keyword*.
- Pemeriksaan token dalam string.

Seperti halnya pengabaian baris dan pernyataan, token yang merupakan bagian dari string tidak akan diperiksa.

Langkah kedua adalah mengambil daftar nama fungsi dalam kode program file input. Daftar nama fungsi tersebut nantinya akan dimasukkan dalam daftar nama fungsi input yang bisa dipilih *user* melalui suatu *combo box.*Fungsi ini diperiksa berdasarkan tipe data kembalian, nama dan parameter. Dengan demikian aplikasi ini mengenali keberadaan *polymorphism*.

Langkah ketiga adalah memasukkan fungsi yang akan diuji coba. Fungsi dapat diperoleh dari daftar nama fungsi yang telah diperoleh pada langkah kedua. Pada langkah ini, isi fungsi akan direpresentasikan dalam bentuk *graph* untuk mencari setiap *path* program dalam fungsi. *Path* di sini diartikan sebagai urut-urutan pernyataan dari suatu node yang merupakan *leaf* sampai pada *root*. *Leaf* adalah suatu node yang tidak mempunyai *child* dan *root* adalah suatu node yang tidak mempunyai *parent*. Representasi *graph* ini dapat divisualisasikan sebagai *control flow graph(CFG).*

3.4. TESTING DATA

Dari representasi *graph* yang dihasilkan dapat diperoleh setiap *path* dalam fungsi yang diinputkan. Setiap pernyataan dalam *path* yang didapatkan akan diambil daftar penggunaan variabelnya atau yang biasa disebut daftar *def-use* variabel. *Def-use* variabel dapat diperoleh dari beberapa proses, misalnya *assignment* (pemerian), persamaan matematika ataupun pemeriksaan kondisi. Pada kondisi awal, semua variabel diberi status *undefined.* Bila suatu variabel diberi suatu nilai (proses *assignment*) maka variabel tersebut dikatakan terdefinisi. Jadi statusnya berubah menjadi *defined*. Status ini pun akan berubah sesuai dengan pernyataan-pernyataan berikutnya. Sebagai contoh, status ini akan berubah menjadi *used* bila variabel tersebut (variabel yang sudah berada dalam status *defined*) diacu oleh variabel lain.

Pencarian *def-use* ini sebenarnya merupakan langkah awal dari pencarian *data flow anomaly* dalam kode program yang diinputkan. Pencarian *data flow anomaly* sendiri dilakukan dengan cara mencari keterkaitan suatu variable dengan daftar variable yang telah teridentifikasi sebelumnya di dalam daftar *def-use* berdasarkan statusnya. Dengan kata lain, setiap variable pada suatu baris akan diperiksa statusnya dan dibandingkan dengan status variable itu sendiri pada baris-baris sebelumnya dan jika tidak ditemukan dicari pada daftar deklarasi konstata dalam file header yang menyertai program yag diuji coba. Karena proses pengujian dilakukan per fungsi maka pencarian *data flow anomaly* juga dilakukan per fungsi dan tidak bersifat *interprocedural*, dalam setiap *path* dalam fungsi yang diuji coba. Proses tersebut dapat dilakukan dengan algoritma sebagai berikut:

```
while(index< size_of_paths) {
  Vector vbes←getVariabelStatusBaris( paths[index]);
  Vector anomalies←new Vector();
  while(i<size_of_vbses){
    j←i-1;
    while(j>0){
      anomaly←check_anomaly(vbses[i], vbses[j]);
      anomalies.addElements(anomaly);
      j--;
    }
    i++;
  }
  index++;
}
```

Dalam algoritma tersebut dijelaskan bahwa setiap *path* dalam fungsi input akan dicari daftar *def-use*-nya dan kemudian dicari daftar anomalinya.

Untuk meyakinkan bahwa setiap *path* yang ditelusuri adalah benar dan mempermudah untuk memengeti aliran program secara visual maka perlu dibangkitkan *control flow graph (CFG).* Setiap *statement* akan diwakili oleh satu node, dari node yang satu ke node yang lain akan dihubungkan oleh suatu *directed edge* sesuai dengan jenis kontrol. *control flow graph* yang dibangkitkan pada aplikasi ini adalah *control flow graph* dari fungsi yang telah diinputkan user, jadi bukan *control flow graph* untuk keseluruhan program. Kode program akan divisualisasikan sesuai dengan notasi-notasi *flow graph*, seperti yang telah diuraikan bagian 2.

Dalam pembuatannya, *control flow graph* tidak tergantung pada keberadaan *data flow anomaly* karena CFG hanya merupakan proses visualisasi dari aliran program. Jadi, keduanya merupakan suatu proses yang terpisah dan tidak saling berkaitan.

4. HASIL UJI COBA

Untuk mengetahui apakah proses berjalan sebagaimana mestinya, maka pada pelaksanaan uji coba ini dilakukan dengan beberapa skenario. Skenario-skenario tersebut adalah:

- Pengujian fungsi sederhana tanpa anomali(skenario I)
- Pengujian fungsi sederhana dengan anomali(skenario II)
- Pengujian fungsi yang melibatkan pemanggilan terhadap fungsi lain(skenario III)
- Pengujian fungsi yang melibatkan percabangan(skenario IV)
- Pengujian fungsi yang melibatkan perulangan(skenario V)

Masing-masing skenario dilakukan untuk memastikan bahwa aplikasi dapat berjalan pada masing-masing kondisi fungsi seperti dalam skenario yang ditetapkan, baik fungsi sederhana tanpa anomali, dengan anomali, yang melibatkan pemanggilan fungsi lain, percabangan maupun perulangan.

Yang dimaksud fungsi sederhana adalah fungsi yang hanya melibatkan pernyataan-pernyataan sederhana, misal cin>>data1. Fungsi yang

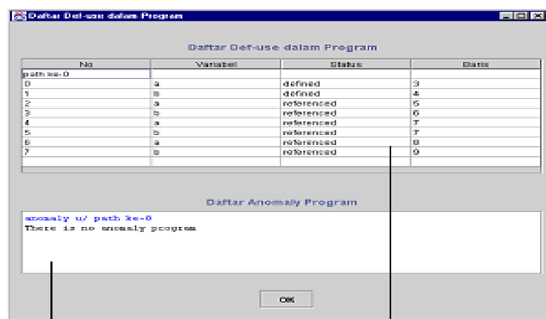
melibatkan pemnggilan terhadap fungsi lain adalah fungsi yang salah satu pernyataan di dalamnya adalah pemanggilan terhadap fungsi lain, misal check_anomaly(vbes[i], vbes[j]). Fungsi yang melibatkan percabangan adalah fungsi yang didalamnya mengandung pernyataan yang mengakibatkan terjadinya percabangan dalam eksekusi program, misalnya pernyataan *if-then-else* dan *switch-case*. Sedangkan fungsi yang melibatkan perulangan adalah fungsi yang didalamnya mengandung pernyataan yang menyebabkan terjadinya perulangan dalam eksekusi program, seperti pernyataan *while-do* dan *for*.

Skenario I

Pada skenario I ini diujicobakan suatu fungsi yang sederhana, yaitu fungsi yang semua pernyataannya berjalan secara sekuensial tanpa percabangan maupun loop. Fungsi input yang dipilih pada skenario ini adalah *void input_tukar ()* dengan kode program sebagai berikut:

```
0:    void input_tukar()
1:    {
2:          int a,b;
3:          a=55;
4:          b=77;
5:          cout<<"main() : a= "<<a<<endl;
6:          cout<<"main() : b= "<<b<<endl;
7:          tukar(a,b);
8:          cout<<"main() : a= "<<a<<endl;
9:          cout<<"main() : b= "<<b<<endl;
10:   }
```

Pada daftar *def-use* gambar 2 tercantum bahwa variabel *a* dan *b* telah didefinisikan (status=*defined*) pada baris ketiga dan keempat sebelum digunakan serta tidak ada pengulangan pendefinisian setelah baris tersebut. Dengan demikian tidak ada *data flow anomaly* dalam fungsi input tersebut. Hasil ini telah sesuai dengan yang diharapkan, yaitu aplikasi dapat membaca baris pernyataan, memberi status tiap variable yang digunakan dalam fungsi serta melakukan pendeksian *data flow anomaly* dengan benar. Dengan demikian dapat disimpulkan bahwa aplikasi ini telah berhasil untuk skenario I, yaitu fungsi sederhana dan tanpa anomali.



Gambar 2. Def-use variabel skenario I

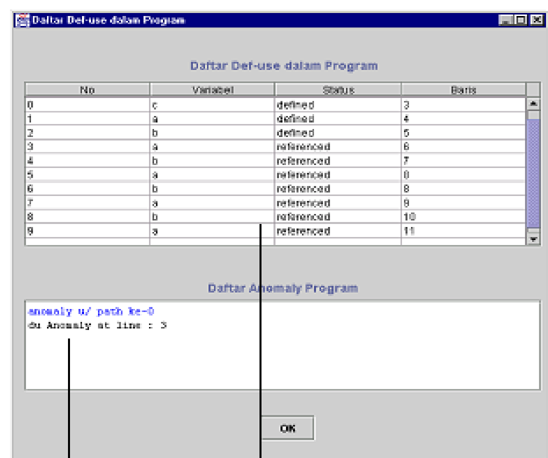
Skenario II

Pada skenario ini akan diujicobakan suatu fungsi yang sederhana, yaitu fungsi yang semua pernyataannya berjalan secara sekuensial tanpa percabangan maupun loop. Fungsi yang akan diinputkan dirancang untuk menghasilkan *data flow anomaly*. Hal ini dilakukan untuk mengetahui apakah perangkat lunak ini mampu mendeteksi keberadaan *data flow anomaly* sesuai yang diharapkan. File input yang diinputkan adalah *void input_tukar2 ( ),* dengan kode program sebagai berikut:

```
0:    void input_tukar2()
1:    {
2:          int a,b;
3:          int c=9;
4:       float d;
5:          a=55;
6:          b=77;
7:          cout<<"main() : a= "<<a<<endl;
8:          cout<<"main() : b= "<<b<<endl;
9:          tukar(a,b);
10:         cout<<"main() : a= "<<a<<endl;
11:         cout<<"main() : b= "<<b<<endl;
12:       cout<<"cek nilai d= "<<d<<endl;
13:   }
```

Pada fungsi di atas digunakan empat variabel, yaitu *a,b,c* dan *d*. Variabel *a* dan *b* telah didefinisikan pada baris kelima dan keenam sebelum diacu pada baris ketujuh sampai kesebelas. Variabel *c* didefinisikan pada baris ketiga dan tidak pernah digunakan. Sedangkan variabel *d* diacu pada baris kedua belas dan belum pernah didefinisikan.

Daftar *def-use* untuk skenario ini dapat dilihat pada gambar 3. Pada daftar *def-use* terlihat bahwa variabel *c* didefinisikan pada baris ketiga dan tidak pernah diacu oleh variabel lain pada baris-baris pernyataan berikutnya sehingga terjadi *du anomaly.* Sedangkan pada baris kedua belas, variabel *d* berstatus *referenced*. Padahal variabel tersebut belum pernah didefinisikan sehingga terjadi *ur anomaly,* yaitu diacu sebelum didefinisikan.



Gambar 3. Def-use variabel skenario II

Skenario III

Pada skenario ini akan diujicoba suatu fungsi yang melibatkan pemanggilan terhadap fungsi lain. Dalam pemanggilan fungsi tersebut, parameter dalam fungsi yang dipanggil akan diberi status *referenced* yang menandakan bahwa variabel tersebut telah diacu atau digunakan dalam fungsi yang memanggilnya. Fungsi yang memenuhi syarat untuk skenario ini adalah fungsi *void input_tukar2().* Dengan demikian, daftar *def-use* maupun *control flow graph(CFG)* yang dihasilkan akan sama dengan yang dihasilkan pada skenario II.

Pada daftar *def-use* variabel *a* dan *b* mempunyai status *referenced.* Dengan demikian aplikasi ini telah mampu mendeteksi adanya pemanggilan fungsi dalam program sehingga parameternya dapat dibaca sebagai variabel dengan status *referenced.*

Skenario IV

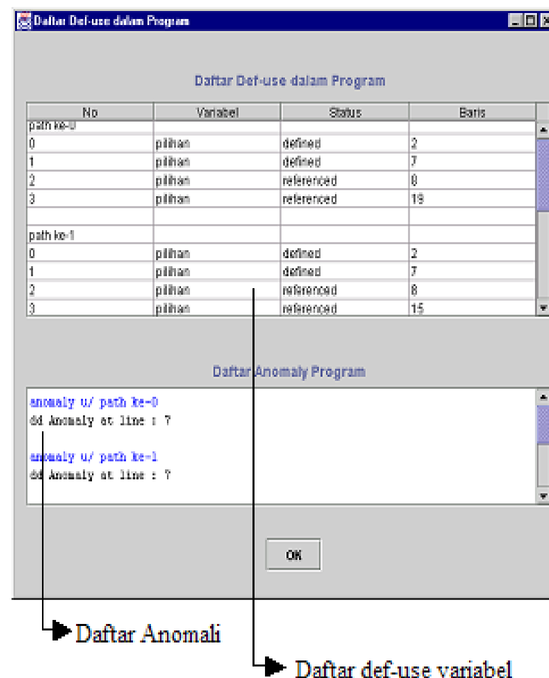
Pada skenario ini akan diujicoba suatu fungsi yang melibatkan percabangan. Hal ini diperlukan untuk memeriksa apakah aplikasi ini mampu mendeteksi keberadaan beberapa *path* dalam fungsi input serta apakah *control flow graph (CFG)* untuk fungsi yang melibatkan percabangan dapat dihasilkan dengan benar.

Salah satu contoh percabangan adalah pernyataan *switch-case*. Pada fungsi yang akan diinputkan, percabangan terjadi pada baris kedelapan dangan percabangan sebanyak empat cabang yaitu pada pernyataan *case1, case2, case3* dan *default*. Variabel yang digunakan adalah *pilihan.* Pada baris kedua, variabel ini telah didefinisikan dan didefinisikan kembali pada baris ketujuh melalui input dari pengguna. Jadi fungsi ini mengandung *dd anomaly* pada baris ketujuh. Fungsi yang diinputkan dalam uji coba ini adalah *void main(),* dengan kode program sebagai berikut:

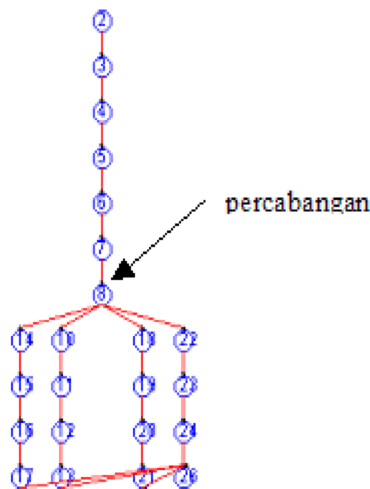
```
0:  void main()
1:  {
2:   int pilihan=0;
3:  cout<<"pilih operasi yang
    diinginkan"<<"\n";
4:  cout<<"1 = jual beli"<<"\t"<<"2 = rata-
    rata"<<endl;
5:  cout<<"3 = membaca tahun"<<"\t"<<"4
    = tukar nilai"<<endl;
6:  cout<<"pilihan :"<<endl;
7:   cin>>pilihan;
8:   switch(pilihan)
9:  {
10:  case 1:
11:  cout<<"pilihan anda : "<<pilihan<<"yaitu
     jual beli"<<endl;
12:  jual_beli();
13:   break;
14:  case 2:
15:  cout<<"pilihan anda :
     "<<pilihan<<"yaitu jual beli"<<endl;
16:  rata_rata();
17:  break;
18:  case 3:
19:  cout<<"pilihan anda : "<<pilihan<<"yaitu
     jual beli"<<endl;
20:  membaca_tahun(2000,2004);
21:  break;
22:  default:
23:        cout<<"pilihan anda :
           "<<pilihan<<"tak ada dalam
           option"<<endl;
24:  printf("Tak ada dalam pilihan\n");
25:  }
26:  getch();
27:  }
```

Pada gambar 4 terlihat bahwa program telah bekerja sebagaimana mestinya, yaitu mampu mendeteksi adanya percabangan dengan memberikan daftar *def-use* variabel untuk masing-masing *path* yang berbeda. Begitu pula dengan daftar anomali. Anomali dideteksi berdasarkan *path* yang dilewati. Misalkan pada *path-1* terjadi anomali pada baris ketujuh karena variabel *pilihan* sudah didefinisikan atau dalam status *defined* pada baris kedua tapi didefinisikan kembali pada baris ketujuh.

Sedangkan *control flow graph (CFG)* untuk fungsi *void main()* dapat dilihat pada gambar 5. Pada gambar tersebut ditunjukkan bahwa perangkat lunak ini telah berhasil membaca percabangan dalam program, yaitu pada baris kedelapan.



Gambar 4. Def-use variabel skenario IV



Gambar 5. Control Flow Graph Skenario IV

5. KESIMPULAN

Adapun kesimpulan dari penelitian ini adalah sebagai berikut:

1. Tanpa benar-benar mengeksekusi program yang diuji coba, dengan metode *static data flow analysis*, daftar penggunaan *(def-use)* variabel dalam program dapat diperoleh.
2. Dengan metode *static data flow analysis,* dapat diketahui keberadaan *data flow anomaly* dalam program pada semua *path* program yang ada. Jadi semua pernyataan program ditelusuri minimal satu kali.
3. *Control Flow Graph (CFG)* dapat dibangkitkan dengan memanfaatkan kedalaman tiap pernyataan dalam program.

6. DAFTAR PUSTAKA

1. W.K. Chan, T.Y. Chen, "An Overview of Integration Testing Techniques for Object-Oriented Programming". 2002. HKU CSIS Tech Report TR-2002-03.
2. Oliver Coul, "White-box Testing". 2003; Available from http://www.ddj.com/documents/s=887/ddj0003a/0003a.htm. Accessed March 19, 2002.
3. Doug Grant, "Project Descriptions". 2002; Available from http://www.it.swin.edu.au/centres/cse/projects.htm. Accessed October 17, 2002.
4. Roger S. Pressman, "Software Engineering: A Practioner's Approach, Fifth Edition". McGraw-Hill. 2001. h. 347-372.
5. Don Lance, Roland H.Untch, Nancy J. Wahl, *"Bytecode-based Java Program Analysis*". 1999.
6. Leon Moonen, "Data Flow Analysis for Reverse Engineering. University of Amsterdam ,Programming Research Group". 1996.
7. T.Y. Chen, C.K. Low, "Dynamic Data Flow Analysis for C++". IEEE. 1995.