Jurnal Ilmiah KURSOR
Menuju Solusi Teknologi Informasi

Vol. 5, No. 2, Juli 2009 **ISSN 0216 - 0544**

VERIFIKASI BERBASIS ASERSI OTOMATIS MODULAR DENGAN METODE *SEPARATION LOGIC*

[*] Luh Putu Ary Tjahyanti, [**] Sarwosri, [***] Bain Khusnul Khotimah
[*] Program Studi Teknik Informatika, AMIKOM-ASM Mataram
Jl.Majapahit Mataram, Nusa Tenggara Barat, 83125
[**] Fakultas Teknologi Informasi, ITS
Jl.Raya ITS, Kampus ITS, Sukolilo, Surabaya, 60111
[***] Jurusan Teknik Informatika, Fakultas Teknik, Universitas Trunojoyo
Jl. Raya Telang PO. BOX 2 Kamal, Bangkalan, Madura, 69162
E-Mail: [*] arytjahyanti@yahoo.com, [**] sri@cs.its.ac.id, [***] bain@trunojoyo.ac.id

Abstract

In doing assertion design program as in the driver of hardware, wrong logic problems is often unpredictable which is caused a complicated listing problems. Proved Mathematics will not solve the problems, but it needs tool detector for wrong logic and programmer who detect the problems. The available compilers verify the code program and warn when mistakes are found. On other hand, listing program having wrong logic often can be compiled, this result in ineffectiveness of programmer's performance. This research tries to offer Accsrtion-Based verification with otomata technique. Accertion- Based Verification covers the specification,design and implementation with separation logic method .Accertion-Based Verification also supports intellegent debug system to help users effectively in understanding the accertain programs and its result. Its principal is that the listing program having Boolean expression will be separated to be verified every variable, modul and used class. This tool is examined with the examples of listing problem having logic debug mistake and listing of the correct logic. The result of toon Accertion- Based Verification may detect the maistake logic and do the process of writing in accertion-based programing.

Key Words: Assertion Based Verification, Separation Logic, Boolean Expression.

PENDAHULUAN

Verifikasi memerlukan hampir 70% dari siklus desain perangkat keras [1]. Kompleksitas rancangan program seperti pembuatan *driver* suatu perangkat keras berimplikasi pada sulitnya melakukan verifikasi secara langsung. *Driver* yang kompleks dapat disebabkan oleh perangkat keras yang besar, tim proyek yang banyak, strategi rancangan dengan performansi tinggi, banyaknya bahasa pemrograman yang dilibatkan dan *tools* pendukung yang digunakan. Kompleksitas ini akan berpengaruh pada kurangnya pemahaman rancangan program, verifikasi yang tertunda atau tidak lengkap, dan siklus *debug* yang cukup panjang.

Untuk mengatasi masalah di atas diperlukan suatu *tool* dan metode verifikasi yang baru. *VBA* atau *Assertion-Based Verification* muncul sebagai metodologi yang ampuh dalam verifikasi desain program [2]. Dengan menggunakan logika sementara, deskripsi yang tepat dari proses yang diharapkan pada desain tersebut dapat dimodelkan, dan deviasi dari proses yang diharapkan ini direkam oleh simulasi atau dengan metode formal [3]. Verifikasi berbasis asersi pada umumnya digunakan dalam pembuktian logika program. Pembuktian logika program tidaklah sama dengan pembuktian teorema matematika. Pembuktian matematika seringkali tanpa petunjuk seperti bagaimana pembuktian itu mungkin dibenarkan. Di sisi lain, tidak dapat

dihindari bahwa banyak *programmer* memiliki pemahaman yang sedikit tentang bagaimana program tersebut semestinya bekerja. Tujuan dari verifikasi program tidaklah mencari pembuktian yang panjang, namun untuk memformalisasi alasan mengapa pada poin dimana terjadi kecacatan menjadi bukti bagi *programmer*.

Reynolds [4] telah menemukan perluasan logika *Hoare* yang disebut logika pemisahan atau "*separation logic*". *Separation logic* adalah logika program yang dilengkapi properti menspesifikasi dan verifikasi dari alokasi-dinamik di-*link*-kan dengan struktur data, yang lebih sederhana dengan spesifikasi dan bukti program daripada formalisme sebelumnya [5]. Asersi-asersinya lebih menjabarkan bentuk struktur data dibandingkan detil kontennya, dan ini menjadi pemikiran untuk membuat *tool VBA* yang dapat berjalan otomatis secara penuh. Simulasi dengan asersi dapat memberikan informasi yang dapat meningkatkan pemahaman kerja internal dari desain serta mengurangi jumlah iterasi.

Tujuan dari penelitian ini adalah untuk membuat spesifikasi, merancang, memvalidasi serta melakukan pengecekan logika dari penggunaan bahasa pemrograman dengan teknik otomata yang menggabungkan berbagai fitur seperti modularitas, *code generation* dan verifikasi otomatis. Untuk kasus dasar ditangani dengan teknik otomata yaitu menuliskan kembali (*rewrite*) *Sequential-Extended Regular Expression* (*SERE*) [6]. Sedangkan untuk kasus logika yang panjang, digunakan integrasi metode *separation logic* dan teknik otomata. *Tool* yang dibuat ini digunakan untuk membantu *programmer* dalam melakukan verifikasi berbasis asersi secara otomatis modular dengan metode *separation logic*.

BAHASA ASERSI DAN PROPERTINYA

Ekspresi-ekspresi *Boolean* di dalam *PSL* dapat menyimbolkan *true* dan *false*. Misalnya ekspresi *Boolean* dinyatakan oleh simbol utama tunggal dilabelkan b_i. Setiap b_i dapat berupa sebuah sinyal tunggal atau sebuah fungsi *Boolean* dari beberapa sinyal. *SERE* digunakan untuk menentukan rantai *event* sementara *Boolean* yang primitif. Contohnya, *SERE* $\{b_1; b_2; b_3\}$ mengevaluasi *true* (adalah cocok, teramati) jika b_3 mengevaluasi *true*, dan dalam *cycle* sebelumnya, b_2 adalah *true*, dan sebelumnya itu, b_1 telah diasersi.

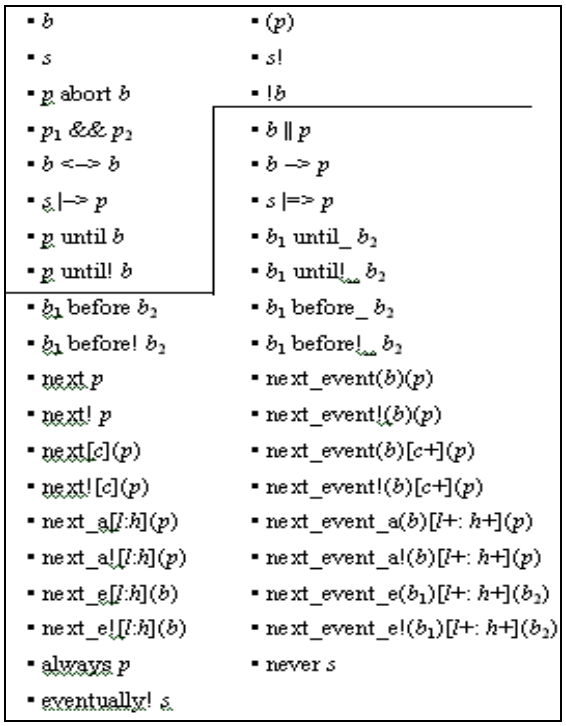
Langkah 1: Jika b adalah ekspresi *Boolean* dan r, r_1 dan r_2 adalah *SERE*, ekspresi berikut adalah *SERE* 1 [4]:

$$\begin{array}{llll} \bullet\, b & \bullet\, \{r\} & \bullet\, r_1; r_2 & \bullet\, r_1 : r_2 \\ \bullet\, r_1 | r_2 & \bullet\, r_1 \,\&\&\, r_2 & \bullet\, [*0] & \bullet\, r[*] \end{array} \quad (1)$$

Dalam konteks asersi, deret dari dua ekspresi *Boolean* b_l; b_r menandakan bahwa ekspresi *Boolean* b_l harus mengevaluasi *true* dalam satu *cycle*, dan b_r harus *true* dalam *cycle* berikutnya. SERE dan deret adalah entitas yang berbeda, dan aturan produksi lebih dipaksakan daripada apa yang dikondisikan dalam langkah *SERE*. *Operator* [∗], | , && dan lainnya dapat dilihat lebih jelas di dalam [6].

Tabel 1. Ekivalensi *Operator PSL SERE* [4].

Ekspresi PSL	Penulisan Kembali SERE
• $r[+]$	$\triangleq$ r ; $r[*]$
• $r[*0]$	$\triangleq$ $[*0]$
• $r[*c^+]$	$\triangleq$ r ; r ; . . . ; r (sebanyak c)
• $r[*l{:}h]$	$\triangleq$ $r[*\mathrm{l}] \mid \ldots \mid r[*\mathrm{h}]$
• $b[{\rightarrow}]$	$\triangleq$ $\{(\sim b)[*]\ ;\ b\}$
• $b[{\rightarrow}\ c^+]$	$\triangleq$ $\{b[{\rightarrow}]\}[*c]$
• $b[{\rightarrow}\ l^+{:}\ h^+]$	$\triangleq$ $\{b[{\rightarrow}]\}[*\mathrm{l}{:}h]$
• $b[= c]$	$\triangleq$ $\{b[{\rightarrow}\ c]\}\ ;\ (\sim b)[*]$
• $b[= \mathrm{l}{:}h]$	$\triangleq$ $\{b[{\rightarrow}\ l{:}h]\}\ ;\ (\sim b)[*]$
• r_1 & r_2	$\triangleq$ $\{\ \{r_1\}$ && $\{r_2; [*]\}\ \} \mid \{\ \{r_1; [*]\}$ && $\{r_2\}\ \}$



Gambar 1. Properti Penulisan Kembali Kasus Dasar dan Ekspresi *Boolean*.

PSL melakukan *sintaks* tambahan *operator* "*sugaring*" yang memudahkan penulisan asersi, tapi tidak menambah kekuatan ekspresif pada bahasanya. *Operator PSL SERE sugaring* yang paling dikenal diperlihatkan pada Tabel 1.

Dari Tabel 1 terlihat *b* adalah ekspresi *Boolean*; *r* adalah sebuah *SERE*; *l*, *h* dan *c* adalah bilangan bulat non negatif dengan $h \geq l$; dan simbol $\triangleq$ menandakan ekivalensi.

Langkah 2: Misalkan *b*, b_1 dan b_2 adalah ekspresi *Boolean*, misalkan *s* adalah sebuah deret dan *p*, p_1 dan p_2 adalah properti. Jika *l*, *h* dan *c* adalah bilangan bulat non negatif dengan $h \geq l$, maka properti bahasa pondasi *PSL* dijabarkan pada Gambar 1 dalam *subset*-nya. Disini $^+$ merupakan bilangan bulat positif.

Properti pada Gambar 1 menunjukkan semantik yang dituliskan kembali atau diimplementasi dalam bentuk *otomaton*. Properti di atas garis batas untuk aturan penulisan kembali pada kasus dasar dan yang di bawah garis akan direncanakan untuk proses *next*. Pada langkah *next* dapat diamati bahwa deret dan ekspresi *Boolean* dapat diterjemahkan menjadi dua *mode* dalam verifikasi dinamik.

Langkah 3: *Mode* kondisional. Konteks dimana deteksi sebuah deret atau ekspresi *Boolean* harus dilakukan. Untuk setiap kondisi mulai dari sebuah ekspresi *Boolean* (deret), sinyal hasil di-*trigger* masing-masing dan setiap waktu ekspresi *Boolean* (rantai *event* dideskripsikan oleh deretnya) diamati.

Langkah 4: *Mode* Obligasi. Konteks dimana kegagalan dari suatu deret atau ekspresi *Boolean* harus diidentifikasi. Untuk setiap kondisi mulai, jika rantai *event*-nya dideskripsikan oleh ekspresi Boolean atau deret tidak terjadi, sinyal hasil akan di-*trigger*. Untuk suatu kondisi mulai dari deret yang diberikan, hanya kegagalan pertama yang diidentifikasi.

Contohnya, dalam asersi

assert never $\{b_1 ; b_2\}$; (2)

assert always $(\{b_1 ; b_2\} \mid\!\!-\!\!> p_1)$; (3)

Deret (2) dan (3) dalam *mode* kondisional karena keberadaannya digunakan untuk mendeteksi kondisi gagal. Di lain pihak, dalam

assert always $\{b_1 ; b_2\}$; (4)

assert always $(b_1 \text{ –> } \{b_2 ; b_3\})$; (5)

Deret (4) dan (5) dalam *mode* obligasi karena kegagalan yang terjadi adalah digunakan untuk men-*trigger* sebuah kondisi. Contoh lain pemakaian properti dalam verifikasi perangkat keras lengkap seperti dalam Persamaan (6).

$$always\left(\{requestA\} \mid\!\!\rightarrow \{(\sim grantA)[*0:15]; grantA\}\right) \quad (6)$$

Properti ini mengkondisikan bahwa sebuah *request* disampaikan ke pemisah, agen *A* akan menerima *grant bust* dalam enam belas *clock cycle*. Jika kondisi mulai tidak terpenuhi, sebuah asersi *error* terjadi.

Otomata untuk Deret dan Ekspresi *Boolean*

Suatu *otomaton* dapat dilukiskan oleh suatu grafik arah, dimana *node* adalah keadaan (*state*), dan kondisi-kondisi untuk transisi di antara keadaan tertulis pada garis hubung seperti yang terlihat pada Gambar 2. Contohnya, suatu *A*(*ekspresi*) dan A_C(*ekspresi*) menandai adanya konstruksi dari suatu *otomaton* untuk ekspresi *PSL* yang ditentukan. Simbol digunakan *s* dan *b* masing-masing untuk mewakili deret dan ekspresi *Boolean*. *A*(*s*) dan *A*(*b*) menandakan *mode* obligasi automata untuk deret dan ekspresi *Boolean*, mengikuti langkah 4. *Mode* ini akan dikerjakan ketika deret atau ekspresi *Boolean* digunakan dalam properti.

Gambar 2a dan 2b menunjukkan otomata sederhana untuk ekspresi *Boolean true* dan *false*, di kedua *mode*. Pada Gambar 2b, karena simbol *false* tidak pernah benar, *otomaton* tidak pernah mencapai keadaan akhir. Gambar 2c

memperlihatkan *mode* kondisional *otomaton* untuk mendeteksi deret {*a* ; *b*[*0:1] ; *c*}. Gambar 2d menunjukkan bagaimana deret yang sama diproses dalam *mode* obligasi oleh suatu *otomaton*. Ketika suatu *otomaton mode* kondisional mencapai keadaan akhir (lingkaran ganda), ekspresi yang mewakili *otomaton* telah dideteksi. Ketika suatu *otomaton mode* obligasi mencapai keadaan akhir, kegagalan pertama dari ekspresi ditemukan. Keadaan dalam lingkaran tebal adalah keadaan mulai dari *otomaton.*

Pemisahan Logika (*Separation Logic*)

Pembacaan spesifikasi pemisahan logika berkaitan dengan memecah tumpukan global (*global heap*) menjadi bagian-bagian kecil yaitu *heaplet* [2]. Asersi mengenai spesifikasi *heaplet* [*P*]*C*[*Q*] berarti jika *C* adalah *heaplet* yang memenuhi *P* maka *C* tidak akan mengakses *heap* di luar *P* (selain sel-sel yang dialokasikan selama eksekusi) dan jika *C* berakhir akan memberi *heaplet* yang memenuhi *heap Q*.
Contoh *pseudo code* prosedur *disposing tree*:

```
disp_tree(p) [tree(p)] {
  local i,j;
  if (p = nil) {} else {
    i := p→l; j := p→r; disp_tree(i);
    disp_tree(j); dispose(p); }
} [emp]
```

Di sini secara *rekursif subtree* kiri dan kanan dibuang (*dispose*) kemudian *pointer* utama. *Precondition* dan *postcondition* berkaitan dengan spesifikasi tersebut:

```
[tree(p)] disp_tree(p) [emp]   (7)
```

Pembuktian secara eksekusi simbolik sebagai berikut:

```
[(p→l: x, r: y) * tree(x) * tree(y)]
i := p→l; j := p→r;
[(p→l: i, r: j) * tree(i) * tree(j)]
disp_tree(i);
[(p→l: i, r: j) * tree(j)]
disp_tree(j);
[p→l: i, r: j]
dispose(p);
[emp]
```

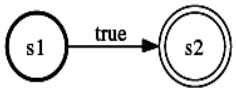
Mentransformasi Properti ke dalam Graf

Di sini ditunjukkan bagaimana mengubah properti ke dalam otomata, untuk konversi yang berikut ke bentuk *graf*. Susunan properti (*p*) hanya melibatkan *otomaton* yang dihasilkan oleh argumentasi properti ini, yakni *A*(*p*) (Persamaan b).

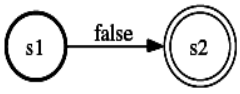
$$A((p)) = A(p) \quad (8)$$

Properti bagaimanapun tak terpisahkan dalam *mode* obligasi. Untuk menyediakan informasi *debugging*, obligasi tidak terbatas pada kegagalan yang pertama untuk masing-masing kondisi mulai. Sebagai contoh, *never* {*a*} dibuat untuk men-*trigger* setiap kali *a* diamati. Karena properti untuk mendeteksi kegagalan, *PSL assert p* tidak mempengaruhi otomata untuk *p*.

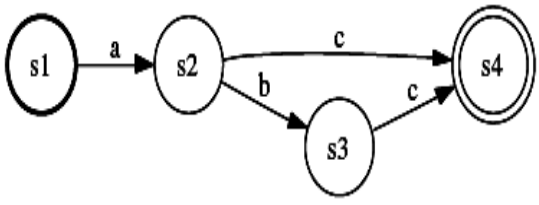
$$A(\textit{assert } p) = A(p) \quad (9)$$



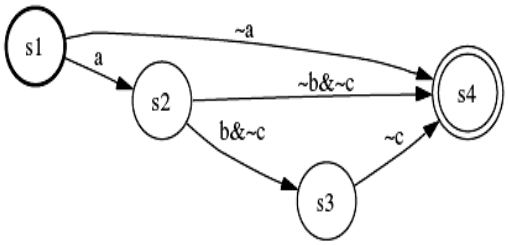
(a) A_C (*true*), *A*(*false*)



(b) A_C (*false*), *A*(*true*)



(c) A_C ({*a* ; *b*[*0:1] ; *c*})



(d) *A*({*a* ; *b*[*0:1] ; *c*})

Gambar 2. *A*(*Ekspresi*) dan A_C(*Ekspresi*) Suatu Otomata untuk *PSL*.

Aturan Rewrite untuk Properti

Di sini akan diperkenalkan kumpulan aturan *rewrite* yang cocok *subset PSL*, dalam konteks

verifikasi dinamik. Langkah *sugaring* berikut contoh aturan *rewrite*-nya:

$$always\ p = \neg\ eventually!\ p\ (Gp = \neg F \neg p) \quad (10)$$

Langkah *sugaring* untuk *always* tidak dapat dipakai dalam *subset* sederhana karena menegasikan sebuah properti tidak diperbolehkan. Disini aturan *rewrite* kompatibel dengan *subset* sederhana yang dikembangkan. Dalam beberapa kasus, cara termudah untuk menangani suatu *operator* adalah *rewrite* menggunakan *operator* yang lebih kompleks. Contohnya, menulis kembali *next_a* memakai *next_event_a* akan terlihat lebih kompleks. Aturan *rewrite* yang dikembangkan dalam *subset PSL* sederhana yang ditunjukkan dalam Persmaam (11) dan (12).

$$\underline{b \,||\, p} \quad \widehat{=} \quad (\sim b) -> p \quad (11)$$

$$\underline{b \ -> p} \quad \widehat{=} \quad \{b\} \,|-> p \quad (12)$$

Karena ekspresi *Boolean* dapat dengan mudah dinyatakan sebagai sebuah deret, kita dapat tulis kembali bentuk di atas ke implikasi sebuah akhiran.

$$always\ p \quad \widehat{=} \quad \{[+]\} \,|-> p \quad (13)$$

$$never\ s \quad \widehat{=} \quad \{[+]: s\} \,|-> false \quad (14)$$

Ketika suatu properti harus selalu *true*, dapat dilihat sebagai *postcondition* dari implikasi akhiran dengan *precondition* terasersi secara konstan ([+] adalah *sugaring* untuk *true*[+]).

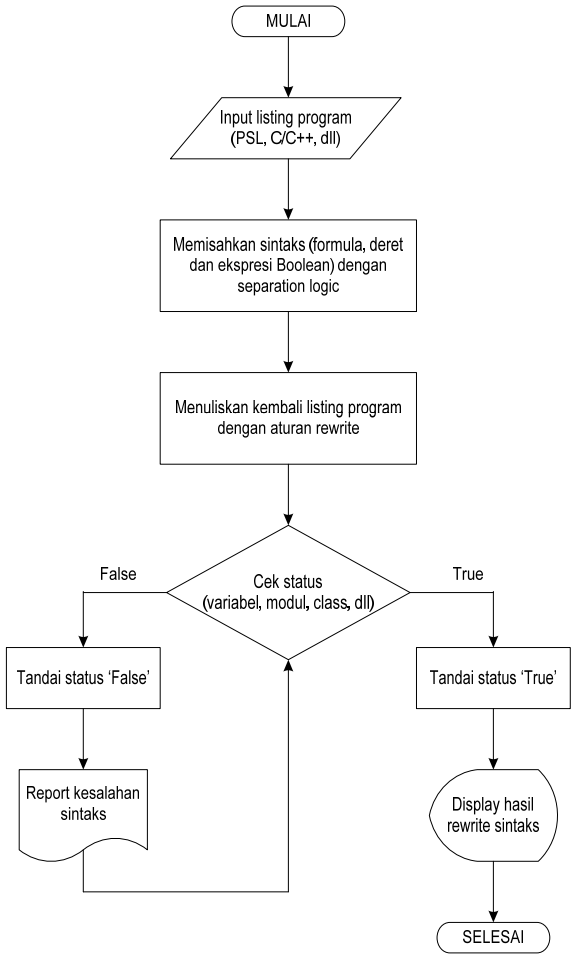
Metode ini cocok untuk efisiensi implementasi *properti PSL* dalam *generator* verifikasi. Kasus dasar ditangani menggunakan otomata untuk *SERE*. Sejumlah aturan *rewrite* mencatat semua *sintaks* dengan teknik otomata dapat diimplementasikan dalam *tool* yang menggunakan *PSL* untuk verifikasi dinamis.

HASIL DAN PEMBAHASAN

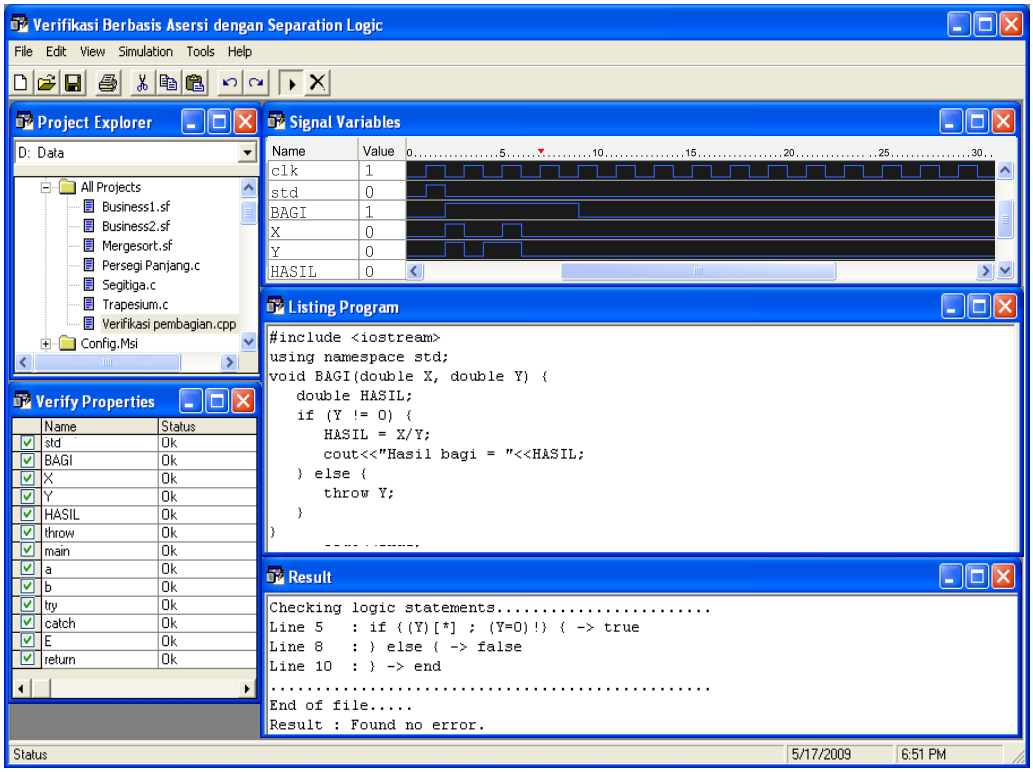
Dalam penelitian ini akan dibangun sebuah sistem yang dapat melakukan verifikasi logika berbasis asersi dengan menggunakan metode *separation logic*. Bentuk diagram alir proses verifikasinya dapat dilihat pada Gambar 3.

Proses verifikasi berbasis asersi di sini terdiri dari beberapa langkah antara lain:

1. Memasukkan *listing program* ke *window Listing Program* dalam *tool VBA*. *Clock cycle* digunakan untuk melakukan pembacaan variabel yang terdapat pada program.
2. Setiap variabel, modul, *class* yang yang akan diuji ditempatkan di dalam properti deret *subsquence* dimana masing-masing variabel yang diuji akan ditandai masuk dalam status ada/tidak dalam memori, apakah bernilai *OK* atau *Failed*.
3. Dengan menggunakan teknik otomata untuk *SERE*, hasil penulisan ulang dengan aturan *rewrite* akan ditampilkan dalam *template* di bawahnya.
4. Setiap koreksi yang dilakukan dalam *listing program* akan meng-*update* Langkah 2 hingga Langkah 4 hingga tercapai *listing program* yang sesuai keinginan *programmer*-nya.



Gambar 3. Diagram Alir Proses Verifikasi.



Gambar 4. Tampilan *Tool VBA* dengan *Separation Logic*.

```
:
:
always @posedge plb_chk or negedge reset
begin
    if (!reset) plb_pavalid = 1'b0;
    else plb_pavalis = pavalid;
end
:
:                       error
```

Gambar 5. Contoh *Listing* Program dengan *Bug*.

```
always @(posedge plb_clk) begin
  a1: assert(!(plb_pavalid && plb_savalid));
      else $error("prim and sec valid at the same time");
  if (plb_savalid && sl_addrack && plb_rnw)
      a2: assert(!plb_rdprim*[0:17];plb_rdprim) @@ (posedge
          plb_clk);
  if (sl_addrack && state == SA_ACK)
      a3: assert((state != LAST)*[0:18];(state == LAST))
          @@(posedge plb_clk);
end
```

Gambar 6. Hasil Penulisan Kembali Bahasa Asersi.

Antarmuka implementasi *software* verifikasi berbasis asersi yang lengkap dengan contoh simulasi program dapat dilihat pada Gambar 4. Contoh *listing program* dengan bahasa Pemrograman *C++* yang diuji pada Gambar 4 tidak terdapat *bug*, sehingga tidak ditemukan *error* pada pemeriksaan pernyataan logikanya.

Salah satu contoh *listing program* yang mengandung *bug* dapat dilihat pada Gambar 5. *Listing program* pada Gambar 5 terdapat satu *bug* yaitu penulisan *plb_pavalis* yang sebenarnya adalah *plb_pavalid*. Hasil penulisan kembali untuk contoh pada Gambar 5 dapat dilihat pada Gambar 6.

Walaupun dalam proyek nyata *bug* seperti ini mungkin telah ditemukan dalam *fase* integrasi sistem, namun cukup mudah memperbaiki programnya saat masalahnya terungkap. Dapat diistilahkan sulit ditemukan namun mudah diperbaiki. Verifikasi seperti ini mendukung sistem *debug* cerdas (*intelligent debug system*) yang diperlukan untuk membantu pengguna secara efektif memahami asersi-nya dan hasil-hasilnya.

SIMPULAN

Dari penelitian ini ada beberapa hal yang dapat disimpulkan, yaitu:

1. Kompleksitas dalam pembuatan desain program seperti *driver* suatu perangkat keras menjadi masalah dalam proses verifikasi dan dapat diselesaikan dengan verifikasi berbasis asersi menggunakan metode *separation logic*.
2. Penerapan metode *rewrite SERE* dilakukan pada asersi yang *error* dan diterjemahkan ke dalam bahasa asersi yang berkaitan baik secara simbolis maupun dengan pengujian *separation logic*.
3. Keuntungan dari verifikasi berbasis asersi yaitu mendeteksi *bug* lebih awal dan men-*debug* inti penyebab permasalahan lebih cepat.

DAFTAR PUSTAKA

[1] Hurd J. *Assertion Based Verification*. Oxford: Oxford University, 2004.

[2] Foster H, Krolnik A and Lacey D. *Assertion-Based Design, 2nd Edition*. London: Kluwer Academic Publisher. 2004.

[3] Berdine J, Calcagno C and Peter W O. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. *FMCO*. 4111: 115-137 2005.

[4] Reynolds JC. *An Overview of Separation Logic*. Pennsylvania: Computer Science Department Carnegie Mellon University. 2005.

[5] Boul´e M and Zilic Z. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. *Computers and Digital Techniques, IET*. 1:669-677. 2006.

[6] Accellera. *Property Specification Languange Reference Manual, v.1.1*. California: Accellera Organization. 2005.