PERANCANGAN FUNGSI RANGKAIAN LOGIKA MELALUI FINITE STATE MACHINE DENGAN MEKANISME BELAJAR DARI CONTOH

Yudhi Purwananto, Chastine Fatichah, Junaidi

Jurusan Teknik Informatika, Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember (ITS) - Surabaya
Kampus ITS, Jl. Raya ITS, Sukolilo – Surabaya 60111
Tel. + 62 31 5939214, Fax + 62 31 5939363
Email : yudhi@its-sby.edu

ABSTRAK

Perancangan rangkaian logika secara umum dapat dilakukan dengan metode sebagai berikut : mendefinisikan permasalahan, membentuk diagram keadaan atau tabel keadaan, membentuk tabel kebenaran, membentuk fungsi logika dan menuangkannya dalam rangkaian logika. Penerapan metode tersebut membutuhkan ketelitian dan waktu yang sebanding dengan kompleksitas rangkaian yang dirancang. Karena itu, perlu dibuat sebuah perangkat lunak untuk membantu proses perancangan rangkaian logika, agar proses perancangan dapat dilakukan dengan mudah, cepat, dan dengan ketelitian yang tinggi.

Untuk merancang rangkaian logika,dalam penelitian ini diterapkan mekanisme belajar dari contoh, yang memanfaatkan teori-teori otomata dan switching. Penelitian ini mencoba membangun model untuk rangkaian logika yang diinginkan berdasarkan atas contoh input/output dari rangkaian logika yang diinginkan, dengan langkah-langkah : membangun model dari contoh yang diberikan, melakukan modifikasi model (jika diperlukan) berdasarkan atas contoh tambahan yang diberikan, memilih salah satu model sebagai solusi akhir jika diperoleh lebih dari satu model, dan menerjemahkan model ke dalam rangkaian logika.

Penelitian ini menghasilkan model untuk rangkaian logika dalam bentuk finite state machine, yang kemudian diterjemahkan menjadi fungsi logika dalam bentuk sum of product. Dari hasil uji coba yang telah dilakukan, untuk setiap contoh yang diberikan, perangkat lunak yang dikembangkan terbukti mampu membentuk minimal sebuah model dan menerjemahkannya menjadi fungsi logika. Namun demikian, tingkat kesesuaian antara model yang dihasilkan dengan yang diharapkan masih tergantung pada kelengkapan contoh yang diberikan. Selain itu, perangkat lunak ini juga mampu membedakan jenis rangkaian logika yang dihasilkan, mampu melakukan modifikasi pada model yang dihasilkan serta mampu menangani contoh-contoh yang inkonsisten.

1. PENDAHULUAN

Untuk merancang sebuah rangkaian logika dengan metode perancangan tradisional, seorang perancang harus melakukan beberapa langkah, yaitu mendefinisikan permasalahan, menggambar diagram keadaan (*state diagram*), menentukan tabel keadaan (*state table*), mereduksi jumlah state pada tabel keadaan menjadi seminimal mungkin, mengkodekan state-state dalam bilangan biner dan mengolahnya menjadi rangkaian logika [10]. Selain itu perancang juga dapat menggunakan metode perancangan terbaru, seperti *hardware definition language (VHDL)*. Dengan metode ini perancang harus menuliskan naskah program untuk mendefinisikan rangkaian logika yang diinginkan [1]. Kedua metode ini mampu menghasilkan rangkaian logika yang cukup akurat. Namun demikian, kedua metode di atas cukup menghabiskan waktu, dan membutuhkan intuisi dan ketelitian yang bagus dari seorang perancang [1].

Karena itu diperlukan suatu perangkat lunak untuk memudahkan dan mempercepat proses perancangan rangkaian logika. Perangkat lunak ini akan menggunakan mekanisme belajar dari contoh (*learning by example*) yang berdasarkan atas teori-teori *switching* dan *otomata*. Dengan menggunakan perangkat lunak ini, perancang cukup mendefinisikan satu set contoh input-output rangkaian logika yang diinginkan, yang terurut berdasarkan waktu, dan kemudian akan mendapat hasil akhir berupa *finite state machine*, yang dapat dengan mudah diterjemahkan ke dalam rangkaian logika, dan diterapkan dalam hardware dengan menggunakan teknologi-teknologi perangkat keras saat ini, seperti *VHDL*. Selain itu perancang dapat memodifikasi rancangan yang dihasilkan dengan cara memberikan contoh-contoh tambahan, untuk mendapatkan rancangan yang lebih spesifik dan sesuai dengan kebutuhan.

2. DEFINISI-DEFINISI

Pada penelitian ini menerima input berupa contoh *input-output* dari sebuah rangkaian logika, yang terurut berdasarkan waku, yang untuk selanjutnya akan disebut sebagai *input/output sequence*. Sedangkan output yang dihasilkan adalah sebuah model dalam bentuk *finite state machine (FSM)* dan fungsi logika yang sesuai untuk model tersebut, dalam bentuk *sum of products(SOP)*.

Definisi 2.1 : Jika *I* adalah sebuah himpunan input yang tidak kosong dan *O* adalah sebuah himpunan output yang tidak kosong. Sebuah *input sequence* didefinisikan sebagai $(x_0,x_1,\ldots,x_m) \in I^m$, sebuah *output sequence* didefinisikan sebagai $(y_0,y_1,\ldots,y_m) \in O^m$ dan sebuah *input/output sequence* didefinisikan sebagai $((x_0,y_0),(x_1,y_1),\ldots,(x_m,y_m)) \in (IxO)^m$ [1].

Sebagai contoh, gambar 1 merupakan contoh *input/output sequence* dari rangkaian *updown counter*.

Step :	0	1	2	3	4	5	6	7	8	9
In :	1	1	1	1	0	0	0	0	0	1
Out :	01	10	11	00	11	10	01	00	11	00

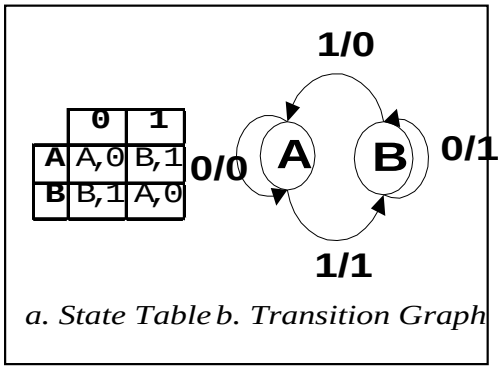
Gambar 1. Contoh Input/Output *Up-down Counter*

Definisi 2.2 : Sebuah FSM adalah merupakan *6-tuple* $M = (I , O , S , s_0 , \delta , \lambda)$, dimana *I,O,* dan *S* adalah himpunan yang tidak kosong, $s_0 \in S$, $\delta: IXS \rightarrow \rho(S)$, dan $\lambda: IXS \rightarrow \rho(O)$.

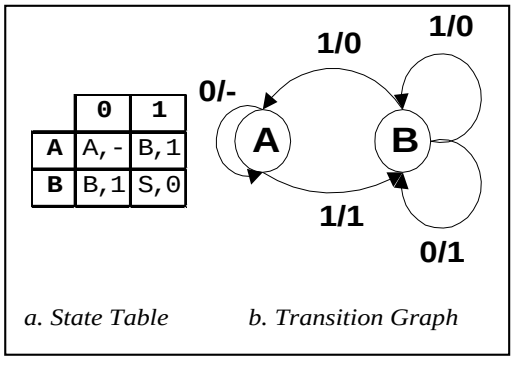
I adalah himpunan symbol-simbol input, O adalah himpunan symbol-simbol output, dan S adalah himpunan symbol-simbol state. s_0 adalah state awal (reset) dari FSM. $\delta : I \, X \, S \rightarrow \rho(S)$ adalah fungsi perpindahan state dimana $\rho(S)=\{A \mid A \subseteq S \}$ yang selanjutnya akan disebut sebagai *next state*. $\lambda : I \, X \, S \rightarrow \rho(O)$ adalah fungsi output dimana $\rho(O)=\{A \mid A \subseteq O \}$.

Sebuah FSM dapat dinyatakan dalam bentuk tabel, yang disebut sebagai tabel keadaan (*state table*), dimana setiap input diwakili oleh kolomnya dan setiap state diwakili oleh barisnya. Selain itu sebuah FSM dapat dinyatakan dalam bentuk graph berarah, yang dikenal sebagai *transition graph*, dimana sebuah state digambarkan sebagai sebuah *node*, sedangkan perubahan statenya digambarkan dengan anak panah berarah (*edge*). Masing-masing *edge* memiliki label berupa pasangan input-output. Kedua jenis cara menyatakan FSM ini, ditunjukkan pada gambar 2.

FSM pada gambar 2, disebut sebagai *completely specified finite state machine (CSFSM)*, karena untuk semua pasangan state $s \in S$ dan input $i \in I$, FSM tidak memiliki output *don't care* dan seluruh perpindahan statenya dispesifikasikan dengan lengkap. Jika sebuah FSM memiliki paling tidak sebuah output *don't care* atau memilliki perpindahan state yang tidak ditentukan dengan lengkap, maka FSM tersebut disebut sebagai *incompletely specified finite state machine (ISFSM)*. Gambar 3 adalah contoh representasi sebuah ISFSM. Pada gambar tersebut terlihat, ketika state *A* mendapat input *0*, FSM akan menghasilkan output *don't care* yang dinotasikan dengan **'-'**. Selain itu ketika state *B* menerima input *1*, terjadi perpindahan state ke state yang tidak didefinisikan dengan lengkap, yang dinotasikan dengan state *S*. State *S* ini yang dapat berupa state apapun dalam FSM, termasuk state saat ini. Karena itu, pada *transition graph*, terlihat state *B* memiliki dua *outgoing edge* dengan label yang sama, tapi menuju *node* yang berbeda.



Gambar 2. Representasi FSM



Gambar 3. Representasi ISFSM

Definisi 2.3 : *Sum Of Products(SOP)* adalah sebuah ekspresi fungsi logika yang didapat dari peng-*OR*-an *term-term AND* yang masing-masing *term* dapat terdiri atas satu atau lebih variabel logika[10].
Contoh : $F_1 = y + xy + x'yz'$.

3. MEMBENTUK MODEL BARU DARI SEBUAH CONTOH

Proses pembentukan model baru terdiri atas dua fase utama, yaitu :

(1) Pembentukan ISFSM dari *input/output sequence*.
(2) Reduksi ISFSM.

Proses pembentukan ISFSM dari *input/output sequence* adalah berdasarkan teorema berikut ini [1].

Teorema 3.1 : Untuk sebuah *input/output sequence* (x_k,y_k), di mana $k = 0,1,\ldots,m$, pasti ada sebuah FSM yang menghasilkan *output sequence* y_k, jika diberikan *input sequence* x_k..
State *0* dianggap sebagai state awal. Mulai dari state *0*, ketika diberikan input x_0, berdasarkan atas definisi di atas, maka FSM menghasilkan output y_0, dan state berpindah ke state *1*. Pada state *1*, ketika diberikan input x_1, ia akan menghasilkan output y_1, dan state berpindah pada state *2*, demikian seterusnya sampai state terakhir *m*.
Sebagai contoh, contoh *input/output sequence* pada gambar 1 dapat dibentuk menjadi ISFSM pada gambar 4.

	1	0
0	**S1,01**	**S,***
1	**S2,10**	**S,***
2	**S3,11**	**S,***
3	**S0,00**	**S,***
4	**S,***	**S3,11**
5	**S,***	**S2,10**
6	**S,***	**S1,01**
7	**S,***	**S0,00**
8	**S,***	**S3,11**
9	**S0,00**	**S,***

Gambar 4. Pembentukan ISFSM dari *Input/Output sequence*

Fase selanjutnya adalah minimalisasi ISFSM yang diperoleh dari fase sebelumnya. Terdapat beberapa algoritma untuk melakukan reduksi ISFSM[2-8]. Sebagian besar algoritma tersebut[3-8] menggunakan pendekatan standar yang berdasarkan atas enumerasi *compatible set*, pembentukan *maximal compatible* dan *prime compatible*, dan pencarian *minimal closed cover*. Pada penelitian ini menggunakan algoritma yang diusulkan dalam [2], yang tidak menggunakan metode standar. Makalah ini hanya akan menjelaskan ide dasar dari algoritma tersebut. Berikut ini adalah beberapa definisi yang akan dipakai dalam menjelaskan algoritma tersebut.

Definisi 3.1 : Sebuah output c adalah ekivalen dengan output d, dinotasikan dengan $c \approx d$, jika c="*" atau d="*" atau c=d. Jika output c adalah sebuah string yang terdiri atas karakter c_p, untuk p=0,1,…m, dan output d adalah sebuah string yang terdiri atas karakter d_q untuk q=0,1,…,n, maka c=d jika m=n dan untuk seluruh p, c_p='X' atau d_p='X' atau $c_p=d_p$, dimana "*" adalah menunjukkan output *don't care*, dan 'X' menunjukkan bit *don't care* .

Definisi 3.2 : Dua buah state s_i dan s_j pada sebuah FSM adalah compatible jika dan hanya jika keduanya menghasilkan output-output yang ekivalen dan menyebabkan perpindahan state pada state-state yang juga compatible, ketika keduanya mendapat input yang sama. State-state yang tidak compatible disebut incompatible. Pasangan state-state yang compatible itu kita sebut sebagai compatible pair. Sedangkan state-state yang incompatible disebut incompatible pair.

Definisi 3.3 : Sebuah himpunan yang beranggotakan state-state dari sebuah FSM adalah *compatible* jika dan hanya jika masing-masing pasangan state dalam himpunan tersebut adalah *compatible pair*. Himpunan itu kita sebut sebagai *compatible set*

Definisi 3.4 : Sebuah himpunan yang terdiri atas state-state, dinotasikan dengan *IS(c,i)* adalah sebuah *implied set* dari sebuah *compatible set c* untuk input *i*, adalah merupakan next state dari setiap state dalam *c* ketika mendapat input *i*.

Definisi 3.5 : Sebuah *class set* dari sebuah *compatible set c*, dinotasikan dengan *CS(c)*, adalah sebuah himpunan yang beranggotakan seluruh *implied set d*, dimana $|d| > 1$, dan $d \not\subset c$, dan $\forall d' \in CS(c), d \not\subset d'$.

Definisi 3.6 : Sebuah himpunan yang terdiri atas satu atau lebih *compatible set c* disebut **closed** jika dan hanya jika seluruh *class set* dari *c* untuk setiap input, terdapat dalam himpunan itu juga.

Definisi 3.7 : Sebuah himpunan yang terdiri atas satu atau lebih *compatible set* disebut **closed cover**, jika dan hanya jika himpunan tersebut *closed*, dan setiap state dalam FSM terkandung oleh paling tidak sebuah *compatible set* dalam himpunan itu.

Semua algoritma untuk mereduksi FSM[2-8] pada dasarnya adalah untuk menemukan sebuah *closed cover* yang paling minimal kardinalitasnya. Metode yang digunakan dalam [2] melakukan pencarian untuk mencoba memberikan sebuah nilai antara *0* sampai *n-1* pada seluruh state dalam ISFSM dan

melakukan *backtack* jika pemberian nilai tersebut melanggar *constraint-constraint* sebagai berikut :

(1) Jika dua state s_i dan s_j dalam FSM adalah *incompatible pair*, maka nilai yang diberikan pada s_i tidak boleh sama dengan s_j.

(2) Jika dua state s_i dan s_j adalah *compatible pair* dan memiliki *implied pair* s_{i+1} dan s_{j+1}, jika nilai yang diberikan pada s_i sama dengan s_j, maka nilai yang diberikan pada s_{i+1} harus sama dengan s_{j+1} .

Nilai *n* adalah adalah pendekatan jumlah state dalam *finite state machine* yang terduksi. Nilai *n* didapat dari jumlah state pada *maximal incompatible* terbesar. *Maximal incompatible* didefinisikan pada definisi 3.8.

Definisi 3.8 : Sebuah *maximal incompatible* adalah sebuah himpunan yang beranggotakan state-state dalam sebuah *finite state machine*, yang seluruh pasangan state dalam himpunan tersebut adalah *incompatible pair*.

Jika pencarian dengan nilai *n* tersebut gagal, maka *n* akan ditambah satu dan pencarian akan diulang dari awal. Penambahan nilai *n* maksimal hingga sama dengan jumlah state dalam *finite state machine* asal. Setelah proses pemberian nilai selesai, maka state-state tersebut akan dikelompokkan berdasarkan nilainya, dan kelompok-kelompok tersebut merupakan *compatible set* pembentuk *minimal closed cover*. Sebagai contoh, pemberian nilai untuk FSM pada gambar 4 yang tidak menyebabkan konflik dapat dilihat pada gambar 5(i), dan menghasilkan sebuah minimal closed cover pada gambar 5(ii), yang kemudian dibentuk menjadi sebuah FSM yang tereduksi, yang ditunjukkan pada gambar 6.

S0=0	**S5=3**
S1=1	**S6=1**
S2=2	**S7=2**
S3=3	**S8=0**
S4=0	**S9=3**

(i)

Compatible Set	Class Set
(S0,S4,S8)	**(S5,S9**
(S1,S6)	
(S2,S7)	
(S3,S5,S9)	

(ii)

Gambar 5. Pembentukan *Minimal Closed Cover*

	1	0
S0	**S1,01**	**S3,11**
S1	**S2,10**	**S2,01**
S2	**S3,11**	**S0,00**
S3	**S0,00**	**S1,10**

Gambar 6. Hasil Reduksi ISFSM

Dari gambar 5(i) terlihat bahwa setiap state diisi nilai dengan hanya sebuah nilai, yang mengakibatkan masing-masing state terkandung dalam hanya satu compatible set. Dalam kasus tertentu, kondisi ini dapat menghasilkan minimal closed cover yang kurang padat, dan menghasilkan FSM yang banyak mengandung output *don't care*, ataupun perpindahan state yang lebih dari satu, sebagaimana ditunjukkan pada gambar 7.

Step	0	1	2	3	4
Input	**0**	**1**	**1**	**1**	**0**
Output	**0**	**0**	**0**	**1**	**0**

(i)

S0=0	**S3=2**
S1=0	**S4=0**
S2=1	

(ii)

Closed Cover	Class Set
S0,S1,S4	
S2	
S3	

(iii)

	0	1
S0	**S0,0**	**S1,0**
S1	**S,***	**S2,0**
S2	**S,***	**S0,1**

(iv)

Gambar 7. Contoh Lain Pembentukan *Minimal Closed Cover*

Gambar 7(i) adalah contoh *input/output sequence* dari sebuah rangkaian logika. Gambar 7(ii) adalah kombinasi pemberian nilai yang tidak melanggar *constraint*. Gambar 7(iii) adalah sebuah *minimal closed cover*. Gambar 7(iv) adalah FSM yang tereduksi. Pada gambar 7(iv) terlihat beberapa output *don't care* dan perpindahan state yang tidak terdefinisi.

S0=0,1,2	**S3=2**
S1=0	**S4=0,1,2**
S2=1	

(i)

Closed Cover
S0,S1,S4
S0,S2,S4
S0,S3,S4

(ii)

	0	1
S0	**S0,0**	**S1,0**
S1	**S0,0**	**S2,0**
S2	**S0,0**	**S,1**

(iii)

Gambar 8. Contoh Optimasi Algoritma Reduksi FSM

Untuk itu algoritma di atas harus dioptimasi agar jumlah output *don't care* dan jumlah perpindahan state yang tidak terdefinisi menjadi seminimal mungkin. Optimasi dilakukan dengan cara mempertimbangkan untuk memberikan nilai-nilai lain pada setiap state, tanpa mengubah komposisi yang telah diperoleh sebelumnya.

Dengan kata lain, setiap state dapat memiliki satu atau lebih nilai. Gambar 8 adalah hasil optimasi yang dilakukan pada kasus dalam gambar 7. Pada gambar 8(i) terlihat bahwa masing-masing state memiliki lebih dari satu nilai, dan terkandung dalam lebih dari satu *compatible set*, yang kemudian menghasilkan FSM dengan jumlah output *don't care* dan perpindahan state yang lebih sedikit.

4. MEMODIFIKASI MODEL DAN PENGUJIAN KONSISTENSI

FSM yang dihasilkan pada proses sebelumnya dapat berupa ISFSM atau CSFSM, yang mungkin kurang cocok dengan model yang diinginkan. Karena itu, perancang dapat memberikan contoh-contoh tambahan agar model menjadi lebih spesifik dan sesuai dengan model yang diinginkan. Sebelum memodifikasi model, contoh-contoh tambahan yang diberikan harus diuji konsistensinya dengan model tersebut. Pengujian konsistensi ini adalah untuk menjamin untuk menjamin agar model yang dimodifikasi, tetap konsisten dengan contoh-contoh sebelumnya. Proses modifikasi model baru, dapat dilakukan dengan cara menambahkan simbol input baru (bila diperlukan), mendefinisikan output-output yang sebelumnya didefinisikan sebagai *don't care*, dan menghapus kemungkinan-kemungkinan *next state* yang lebih dari satu atau tidak terdefinisi. Proses ini selengkapnya dapat dilihat pada gambar 4.1.

4.1. Pengujian Konsistensi

Untuk mengetahui apa yang akan dilakukan dengan contoh-contoh baru, kita harus memiliki sebuah metode untuk membandingkan contoh-contoh baru tersebut dengan model-model yang telah ada. Metode pengujian konsistensi ini, didasarkan atas definisi berikut[1].

Definisi 4.1 : Sebuah FSM adalah *konsisten* dengan sebuah *input/output sequence* jika sehubungan dengan *input sequence* yang diberikan, FSM menghasilkan paling tidak sebuah *output sequence* yang ekivalen dengan *output sequence* yang diberikan, dan tidak menghasilkan satupun *output sequence* yang tidak ekivalen dengan *output sequence* yang diberikan.

Definisi 4.2 : Sebuah *output sequence p(k)*, untuk *k=0,1,2,…,m* adalah *ekivalen* dengan *output sequence q(t)* untuk *t=0,1,2,..,n*, jika *m=n*, dan untuk *k=0,1,…m*, maing-masing output dalam *p(k)* ekivalen dengan output dalam *q(k)*. Definisi ekivalensi antara dua output dapat dilihat pada definisi 3.1 dalam bagian sebelumnya.

Ketika mendapat *input sequence*, FSM akan menghasilkan satu atau lebih *output sequence*. Dalam proses ini, sistem akan membandingkan *output sequence* yang diberikan dengan yang dihasilkan oleh FSM. Jika FSM menghasilkan paling tidak sebuah *output sequence* yang ekivalen dengan *output sequence* yang diberikan, maka contoh tambahan tersebut adalah contoh yang konsisten, dan model dapat dimodifikasi. Jika selain itu, maka contoh tambahan tersebut adalah contoh yang inkonsisten, sehingga proses modifikasi tidak dapat dilakukan. Untuk menampung informasi-informasi yang inkonsisten, sistem akan mencoba membangun sebuah model baru untuk contoh tersebut. Sehingga, dalam proses perancangan sistem mungkin menghasilkan lebih dari sebuah model.

4.2. Penambahan Simbol-simbol Input Baru

Agar model yang dibangun dapat memperlebar ruang lingkupnya secara luwes, atau untuk menampung contoh-contoh tambahan, sebuah simbol input akan ditambahkan pada himpunan input I dalam FSM, jika input tersebut terdapat dalam *input/output sequence* tapi tidak ada dalam FSM.

S/I	0	1
0	1,1?	1,00
1	0,10	0,01

(NS,Output)

(i)

Input Diberikan	Sequence (a)	Sequence (b)	Output Dibutuhkan
0	0,1?	=	11
n	1,01	=	1
1	0,00	1,01	?1

(Current State,Output)

(ii)

S/I	0	1	n
0	1,1?	1,00	S,*
1	0,10	0,01	1,01

(NS,Output)

(iii)

Gambar 9. Contoh Proses Penambahan Simbol Input

Proses penambahan input baru digambarkan dalam gambar 9. Contoh ini juga menjelaskan bagaimana sistem melakukan pengujian konsistensi.

Sebuah *input/output sequence (0/11,n/01,1/?1)* akan dibandingkan dengan sebuah FSM 9(i). Hasil dari masing-masing pembandingan ini ditunjukkan pada gambar 9(ii). Sedangkan hasil FSM yang dimodifikasi ditunjukkan pada gambar 9(iii).

Langkah-langkah yang dilakukan adalah sebagai berikut :

- Mulai dari state 0, FSM menerima sebuah input 0, dan menghasilkan output 1? (ditunjukkan dalam *sequence* a dalam gambar 9(ii), yang dibandingkan dengan output yang dibutuhkan yaitu 11. Agar kedua output ekivalen, maka kita memberikan nilai 1 pada karakter *don't care*, sehingga output sekarang menjadi 11. State saat ini berpindah ke state 1.

- Pada state 1, FSM tersebut menerima sebuah input n yang tidak ada dalam himpunan input I dari FSM. Sebuah kolom input baru ditambahkan pada *state table*, dan dinamai dengan n (ditunjukkan pada gambar 9(iii). Seluruh entri dalam kolom yang baru diinisialisasi dengan output *don't care*, dan perpindahan statenya diinisialisasi dengan S, yang disebut sebagai *unspecified next state*. Entri pada state saat ini, yaitu state 1, diset menjadi "S,01", karena pada state saat ini FSM, harus menghasilkan output 01, sesuai dengan ouput pada *input/output sequence* yang diberikan. State S dapat bernilai state apa aja dalam FSM , termasuk state saat ini. Jadi S dapat bernilai 0 atau 1. Dengan kata lain, pada state saat ini, FSM memiliki dua kemungkinan perpindahan state. State 0 akan dicek terlebih dahulu, sedangkan state 1 disimpan dalam stack untuk dicek di kemudian waktu. Saat ini state berpindah pada state 0.
- Pada state 0, FSM menerima input 1, dan ia menghasilkan output 00, yang dibandingkan dengan output yang dibutuhkan yaitu ?1. Karakter pertama kedua output itu dapat menjadi ekivalen jika karakter *don't care* diberi nilai 0. Namun karakter kedua tidak dapat dijadikan ekivalen. Jadi, *output sequence* yang baru dihasilkan (ditunjukkan pada sequence a dalam gambar 5(ii) adalah tidak ekivalen dengan *output sequence* yang dibutuhkan. Jadi, output dikembalikan pada ?1, dan output yang baru dihasilkan harus di hapus. Untuk melakukan ihal itu, kemungkinan next state 0 yang dihasilkan pada langkah 2 harus dihapus.
- Untuk menghasilkan *sequence* b, sistem melakukan *backtracking* dan menemukan bahwa pada langkah 2, ada sebuah perpindahan state yang harus dicek, yaitu state 1. FSM sekarang menuju ke state 1 yang menerima input 1 dan menghasilkan output 01, yang dibandingkan dengan output yang dibutuhkan ?1. Dua output ini adalah ekivalen dan output yang dibutuhkan saat ini diset ke 01. Hal ini memperlengkap *sequence* b. Dalam gambar tersebut, tanda = menunjukkan bahwa entri itu sama dengan pada bagian kiri. Untuk melengkapi *sequence* b, sistem tidak perlu membentuk ulang nilai-nilai yang sama yang telah dihasilkan dalam *sequence* a. *Sequence* b, adalah ekivalen dengan *output sequence* yang dibutuhkan. Jadi, perpindahan state 1 tetap dipakai. Langkah ini mengakhiri proses penelusuran secara keseluruhan dan hasil akhir ditunjukkan pada gambar 9.(iii).

FSM yang dihasilkan memiliki sebuah simbol input baru. Karakter output *don't care* dalam FSM asal sekarang memiliki nilai 1.

4.3. Pendefinisian Output-output *Don't care*

Ketika sebuah contoh baru berisi informasi baru yang lebih spesifik, informasi ini akan dimasukkan kedalam model yang telah ada. Sebuah cara untuk melakukan hal ini adalah mendefinisikan output-output yang sebelumnya didefinisikan sebagai output *don't care*.

Untuk memasukkan informasi baru ke dalam sebuah model yang sudah ada sistem hahus membandingkan informasi baru ini dengan informasi yang terkandung dalam model. Dalam tugas akhir ini, proses tersebut dilakukan secara parsial dengan cara membandingkan *output sequence* yang diberikan dengan *output sequence* yang dihasilkan oleh FSM. Proses membandingkan dua *output sequence*, dimana salah satu output mungkin berisi karakter *don't care*, membutuhkan sebuah metode untuk membandingkan dua output dan sebuah metode untuk mencatat karakter-karakter output *don't care* yang telah didefinisikan.

```
Sequence        O   (o1,o2,o3,o4)
                O = (a, c, b, c)
Nilai awal : a=11?, b=?10 dan c="*"

Sequence        P   (p1,p2,p3,p4)
                P = (x, y, x, y)
Nilai awal : x=?10 dan y=1?0

o1 : a=11?,p1 : x=?10 → a=110, x=110
o2 : c= * ,p2 : y=1?0 → c=1?0, y=1?0
o3 : b=?10,p3 : x=110 → b=110, x=110
o4 : c=1?0,p4 : x=1?0 → c=1?0, x=1?0
```

Gambar 10. Contoh Perbandingan Dua *Output sequence*

Sebuah contoh ditunjukkan pada gambar 10 untuk menggambarkan metode yang dipakai. Ada dua *output sequence O* (o_1,o_2,o_3,o_4) dan *P* (p_1,p_2,p_3,p_4). Sequence *O* memiliki 3 nilai yang

berbeda, yaitu *a,b*, dan *c*. Sequence *O = (a,c,b,c)*. Nilai awal unutuk masing-masing output adalah *a=11?*, *b=?10* dan *c=”*”*, dimana *“*”* menunjukkan output *don’t care* dan *‘?’* menunjukkan karakter *don’t care*. Sequence *P* hanya mengandung dua nilai yang berbeda, yaitu *x=?10* dan *y=1?0*. Sequence *P = (x,y,x,y)*.

Proses perbandingan antara dua *output sequence* pada gambar 10. adalah sebagai berikut :

- Perbadingan dimulai dari pasangan yang pertama o_1(a=11?) dengan p_1 (x=?10). Kedua output itu ekivalen, dan nilai kedua output sekarang berubah menjadi a=110 dan x=110. Karakter *don’t care* pada a diisi dengan nilai 0, dan karakter *don’t care* pada x diisi dengan nilai 1.
- Perbandingan selanjutnya antara o_2(c=”*”) dan p_2(y=1?0). Kedua output itu ekivalen, dan nilai keduanya berubah, menjadi c=1?0 dan y=1?0.
- Perbandingan selanjutnya adalah o_3(b=?10) dengan p_3 (x=110), dimana nilai x, adalah hasil dari perbadingan pertama. Keduanya ekivalen, dan keduanya menjadi bernilai b=110 dan x=110.
- Perbandingan terakhir o_4 (c=1?0) dengan o_4(y=1?0). Keduanya ekivalen dan nilai keduanya tidak berubah.

Karena seluruh pasangan yang berkoresponden pada *sequence P* dan *O* adalah ekivalen, maka kedua *sequence* ini ekivalen.

4.4. Penghapusan Perpindahan State yang Non-Deterministic

Cara lain untuk menggabungkan informasi baru pada model yang telah ada dan dengan cara menghapus kemungkinan perpindahan state yang menuju lebih dari satu state dalam FSM. Informasi baru tersebut membantu sistem untuk membedakan kemungkinan-kemungkinan yang akan datang, untuk membatasi pilihan-pilihan atas banyak kemungkinan, atau untuk menghapus *non deterministic* state. Hal ini akan semakin memperjelas lingkup dari model tersebut. Model menjadi lebih spesifik dan terdefinisi dengan lebih baik.

Penghapusan *next state* yang berkoresponden dengan sebuah *output sequence* akan menyebabkan penghapusan *output sequence* tersebut. Agar menghasilkan sebuah FSM termodifikasi yang konsisten dengan *input/output sequence* yang diberikan, *output sequence* yang dihasilkan oleh FSM akan dihapus jika ia tidak ekivalen dengan *output sequence* yang diberikan.

Gambar 11. adalah sebuah contoh untuk memperjelas metode yang dipakai dalam tugas akhir ini. Sebuah *input/output sequence* akan dibandingkan dengan FSM yang didefinisikan pada gambar 11(i). Sehubungan dengan *input sequence* tersebut, FSM akan menghasilkan tiga *output sequence*, seperti yang ditunjukkan pada gambar 11(ii) Perbandingan *output sequence* yang dibutuhkan *(L,L,N)* dengan tiga *output sequence* yang dihasilkan, menunjukkan bahwa *output sequence (a)* adalah ekivalen, *(b)* dan *(c)* tidak ekivalen. Kedua *output sequence (b)* dan *(c)* harus dihapus untuk menghasilkan sebuah FSM yang konsisten dengan *input/output sequence* yang diberikan.

PS/Inp.	X	Y	z
0	**0,1,L**	**1,L**	**0,M**
1	**0,K**	**0,2,L**	**1,N**
2	**1,M**	**0,K**	**0,M**

(Next State, Output)

(i) State Table

Inputs	(a)	(b)	(c)	Outputs
x	**0,L**	**0,L**	**0,L**	**L**
y	**0,L**	**1,L**	**1,L**	**L**
z	**1,N**	**0,M**	**2,M**	**N**

(Present State, Output)

(ii) Sequence yang dihasilkan

Gambar 11. Contoh Pemilihan *Next State* yang Akan Dihapus

Ada tiga kemungkinan *next state* yang bertanggung jawab menghasilkan *sequence (b)*, yaitu *next state* 1 pada state 0 dan input x, dan *next state* 0 pada state 1 dan input y. Penghapusan salah satu *next state* yang manapun dan dua kemungkinan *next state* itu, akan menyebabkan penghapusan *output sequence* yang dihasilkan. Dengan cara yang sama, diketahui bahwa ada dua kemungkinan *next state* yang menghasilkan *sequence (b)*, yaitu *next state* 2 pada state 1 dan input y dan *next state* 1 pada state 0 dan input x. Penghapusan salah satu kemungikinan *next state* ini, akan menyebabkan penghapusan *sequence* (c).

Untuk menghapus *sequence (b)* dan *(c)*, terdapat dua pilihan, yaitu menghapus *next state* 1 pada state 0 dan input x , yang bertanggung jawab dalam menghasilkan dua *output sequence*, atau menghapus kedua *next state* 0 dan 2 pada state 1 dan input y. Jika pilihan pertama dipilih, maka FSM yang dihasilkan akan konsisten dengan *input/output*

sequence yang diberikan. Jika yang dipilih adalah pilihan 2, FSM yang dihasilkan menjadi tidak konsisten dengan *input/output sequence* yang diberikan.

Aturan umum dalam menghapus kemungkinan *next state* adalah :

- Hapus pertama kali, next state yang bertanggung jawab atas sesedikit mungkin *output sequence*.
- Jika hasil dari langkah di atas, ternyata seluruh kemungkinan *next state* untuk masing-masing state dan input terhapus, maka kembalikan seluruh *next state* yang dihapus, dan kemudian hapus *next state* yang bertanggung jawab atas keseluruhan *output sequence*.

5. MEMILIH SALAH SATU MODEL SEBAGAI SOLUSI AKHIR

Dengan pemberian beberapa contoh tambahan, baik yang konsisten ataupun yang inkonsisten, sistem ini mungkin menghasilkan lebih dari sebuah model. Karena itu perlu mekanisme khusus untuk memilih salah satu model yang solusi akhir untuk rangkaian logika yang diinginkan. Mekanisme yang digunakan dalam penelitian ini adalah menggunakan faktor bobot yang disebut sebagai *weight*. Setiap model memiliki weight, dengan aturan pemberian *weight* adalah sebagai berikut :

(1) Model yang pertama kali dibuat, memiliki *weight* sebesar 1.
(2) Jika sebuah model konsisten dengan sebuah contoh tambahan, *weight*nya ditambah 1.

Jika frekuensi munculnya infomasi yang inkonsisten lebih sedikit dari infomasi yang konsisten, maka akan menghasilkan model dengan *weight* lebih tinggi dari model-model yang lain Sistem akan memilih model dengan *weight* tertinggi tersebut sebagai solusi akhir untuk rangkaian logika yang diinginkan.

6. MEMBENTUK FUNGSI LOGIKA DARI MODEL YANG DIHASILKAN

Pembentukan fungsi logika adalah diluar ruang lingkup mekanisme belajar dari contoh. Mekanisme belajar dari contoh berakhir pada proses pemilihan model sebagai solusi akhir. Pembentukan fungsi logika ini akan menggunakan metode perancangan rangkaian logika yang lazim digunakan dalam komunitas ilmu komputer[10].

Pada dasarnya rangkaian logika dapat dikelompokkan menjadi 2 [10], yaitu :

- Rangkaian Kombinasional, yaitu rangkaian logika, yang outputnya, hanya tergantung pada inputnya. Dengan kata lain, rangkaian kombinasional, akan mengeluarkan output yang sama, jika diberikan input yang sama
- Rangkaian Sekuensial, yaitu rangkaian logika, yang outputnya tidak hanya bergantung pada inputnya, melainkan juga pada keadaan (state) rangkaian itu sebelumnya. Rangkaian sekuensial mungkin akan mengeluarkan output yang berbeda ketika mendapat input yang sama.

Setelah sebuah model dihasilkan, sistem akan mengidentifikasi apakah model yang dihasilkan adalah model untuk rangkaian kombinasional atau rangkaian sekuensial. Pengidentifikasi ini didasarkan atas jumlah state dalam FSM. Jika jumlah statenya adalah 1, atau dengan kata lain, tidak ada proses perpindahan state dalam FSM tersebut, maka model tersebut adalah model untuk rangkaian kombinasional. Jika jumlah statenya lebih dari satu, maka model tersebut adalah model untuk rangkaian sekuensial.

Jika model yang dihasilkan adalah model untuk rangkaian kombinasional, maka sistem akan membangun sebuah tabel kebenaran yang ekivalen dengan model tersebut, dengan asumsi output-output yang tidak terdefinisi adalah output *don't care*. Tabel kebenaran tersebut kemudian disederhanakan dengan metode Quine-McCluskey, dan hasilnya berupa fungsi logika dalam bentuk SOP akan diinformasikan pada perancang.

Jika model yang dihasilkan adalah model untuk rangkaian sekuensial, maka sistem akan menggunakan metode perancangan rangkaian sekuensial, dengan langkah sebagai berikut :

- Mengkodekan state-state dalam FSM menjadi bilangan biner (*state assignment*). Tugas akhir ini tidak mencakup algoritma untuk pengkodean state yang paling optimal, karena itu pengkodean state ini diserahkan pada perancang. Secara *default*, pengkodean yang dipakai dalam tugas akhir, adalah menggunakan konversi bilangan desimal ke bilangan biner. Sebelum mengkodekan, sistem akan mendefinisikan jumlah bit bilangan biner yang akan dipakai sehingga dapat mencakup seluruh state. Sebagai contoh, misal sebuah model memiliki 3 state, maka jumlah bit yang dibutuhkan adalah 2, state 0 dikodekan menjadi 00, state 1 menjadi 01, dan state 2 menjadi 10.
- Mengkonversi perpindahan state pada FSM menjadi state yang sesuai dengan hasil proses pengkodean. Bila sebuah state memiliki kemungkinan perpindahan state yang lebih dari satu, maka perpindahan state yang dipakai hanya perpindahan state yang pertama. State-state(dalam bilangan biner) yang tidak terpakai akan diarahkan pada state reset (00).

- Menentukan jumlah dan jenis flip-flop yang akan digunakan.
- Membentuk tabel kebenaran berdasarkan tabel eksitasi dari flip-flop yang dipakai dan menyederhanakan dengan metode Quine-McCluskey.

Hasil akhir dari perancangan rangkaian kombinasional ini adalah fungsi-fungsi logika rangkaian kombinasional dalam bentuk *sum of product*, yang digunakan sebagai output rangkaian logika dan input untuk setiap *flip-flop*.

7. UJI COBA

Uji coba perangkat lunak ini dilakukan pada sebuah sistem komputer dengan spesifikasi prosesor Pentium 200 Mhz, memory 64 MB, hard disk 4 GB, dan Sistem Operasi Microsoft™ Windows 98. Uji coba yang dilakukan meliputi rangkaian kombinasional dan rangkaian sekuensial.

7.1. Uji Coba Rangkaian Kombinasional

Rangkaian kombinasional yang diujikan dalam sistem ini adalah sebagai berikut :

- *Half Adder*
  Sebuah half adder merupakan sebuah rangkaian logika yang melakukan penjumlahan dua buah bilangan biner 2 bit. Hasil penjumlahan berupa bilangan biner 2 bit dan 1 bit sebagai *carry*.

```
Contoh Input/output sequence :
Step :   0      1    2    3    4
In :     1111   0000 0001 0010 0011
Out :    110    000  001  010  011

Step :   5      6    7    8    9
In :     0100   0101 0110 0111 1000
Out :    001    010  011  100  010

Step :   10     11   12   13   14   15
In :     1001   1010 1011 1100 1101 1110
Output : 011    100  101  011  100  101

State Table :
    1111   0000   0001   0010   0011
S0  S0;110 S0;000 S0;001 S0;010 S0;011

    0100   0101   0110   0111   1000
S0  S0;001 S0;010 S0;011 S0;100 S0;010

    1001   1010   1011   1100   1101   1110
S0  S0;011 S0;100 S0;101 S0;011 S0;100
    S0;101

Fungsi Logika :
Y0 = X1X3 + X0X2X3 + X0X1X2
Y1 = X0'X1'X3 + X0'X1X3' + X1'X2'X3 +
     X0X1'X2X3' + X1X2'X3' + X0X1X2X3
Y2 = X0'X2 + X0X2'
```

Gambar 12. Uji Coba Rangkaian Half Adder

  Proses pembentukan model half adder ditunjukkan pada gambar 12.
- *Decimal To Excess-3 Converter*
  Rangkaian logika ini akan menerjemahan bilangan desimal (0,1,…,9() ke kode *Excess-3*. Proses pembentukan model untuk rangkaian logika ini ditunjukkan pada gambar 13.

```
Contoh input/output sequence :
Step:   0      1      2      3      4
In :    0000   0001   0010   0011   0100
Out :   0011   0100   0101   0110   0111

Step:   5      6      7      8      9
In :    0101   0110   0111   1000   1001
Out :   1000   1001   1010   1011   1100

State Table :
    0000    0001    0010    0011    0100
S0  S0;0011 S0;0100 S0;0101 S0;0110 S0;0111

    0101    0110    0111    1000    1001
S0  S0;1000 S0;1001 S0;1010 S0;1011 S0;1100

Fungsi Logika :
Y0 = X3 + X1X2 + X0X2
Y1 = X0'X1'X2 + X1X2' + X0X2'
Y2 = X0'X1' + X0X1
Y3 = X0'
```

Gambar 13. Uji Coba Rangkaian *Decimal To Excess-3 Converter*

- *Odd Parity Generator*
  Rangkaian logika ini akan membangkitkan sebuah bit tambahan pada sebuah data serial agar jumlah bit yang bernilai 1 menjadi ganjil. Proses pembentukan model untuk rangkaian logika ini ditunjukkan pada gambar 14.

```
Contoh Input/Output Sequence :
Step :   0   1   2   3   4   5   6   7
In :     000 001 010 011 100 101 110 111
Out :    1   0   0   1   0   1   1   0

State Table :
    000   001   010   011   100   101   110   111
S0  S0;1  S0;0  S0;0  S0;1  S0;0  S0;1  S0;1
    S0;0

Fungsi Logika :
Y0 = X0'X1'X2' + X0'X1X2 + X0X1'X2 + X0X1X2'
```

Gambar 14. Uji Coba Rangkaian *Odd Parity Generator*

7.2. Uji Coba Rangkaian Sekuensial

Rangkaian sekuensial yang akan di ujikan pada perangkat lunak ini, adalah sebagai berikut:

- *Sequential Parity Generator*
  Rangkaian logika ini didefinisikan dengan contoh *input/output sequence* yang dituinjukkan pada gambar 7.4. Pada langkah ke-k, rangkaian

logika ini harus menghasilkan output 1, jika sampai langkah ke k terdapat sejumlah ganjil bit 1, dan menghasilkan output 0 jika terdapat sejumlah genap bit 1. Model yang dihasilkan serta fungsi logika yang dihasilkan juga ditunjukkan pada gambar 15.

```
Contoh Input/Output Sequence :
Step :    0 1 2 3 4 5 6 7 8 9 10
In :      0 1 0 0 1 0 1 0 1 1 0
Out :     0 1 1 1 0 0 1 1 0 1 1

State Table :
        0               1
S0      S0;0            S1;1
S1      S1;1            S0;0

Fungsi Logika :
FF Count : 1                Use JK Flip-Flop
Output : Y0 = Q0'X0 +       J0 = X0
Q0X0'                       K0 = X0
Flip-Flop Input :
Use RS Flip-Flop            Use T Flip-Flop
S0 = Q0'X0                  T0 = X0
R0 = Q0X0
                            Use D Flip-Flop
                            D0 = Q0'X0
```

Gambar 15. Uji Coba *Sequential Parity Generator*

- *Up-down Counter*
  Rangkaian ini membutuhkan sebuah dengan sebuah input, dimana jika input tersebut bernilai 1 maka *counter* tersebut berfungsi sebagai *up counter* yang menghitung naik, 0,1,2,3,0,dan seterusnya . Namun jika input bernilai 0, maka *counter* akan berfungsi sebagai *down counter*, yang menghitung mundur 0,3,2,1,0,dan seterusnya. Proses pembentukan model untuk rangkaian ini ditunjukkan pada gambar 16.

```
Contoh Input/Output Sequence 1 :
Step :  0   1   2   3   4   5   6   7   8   9
In :    1   1   1   1   0   0   0   0   0   1
Out :   01  10  11  00  11  10  01  00  11  00

State Table :
        1               0
S0      S1;01           S3;11
S1      S2;10           S2;01
S2      S3;11           S0;00
S3      S0;00           S1;10
Weight : 1
```

Gambar 16. Uji Coba *Up-Down Counter*

Untuk memastikan apakah model yang dihasilkan sesuai dengan rangkaian logika *up-down counter*, akan diberikan contoh tambahan, dan hasil prosesnya ditunjukkan pada gambar 17.

```
Contoh Input/Output Sequence Tambahan :
Step :  0   1   2   3   4   5   6   7   8   9   10
In :    0   1   1   0   1   1   0   1   1   0   1
Out :   11  00  01  00  01  10  01  10  11  10  11

Checking Consistency And Updating Model(s)
Example 'CounterEx1' is NOT CONSISTENT with
Model 'CounterModel0'
Example 'CounterEx1' is NOT CONSISTENT with Any
Model
Create New Model From Example 'CounterEx1'

State Table :
        0               1
S0      S1;11           S2;01
S1      S3;10           S0;00
S2      S0;00           S3;10
S3      S2;01           S1;11
Weight : 1

Checking Consistency With Previous Example(s)
Example 'CounterEx0' is CONSISTENT with Model
Weight : 2

Fungsi Logika :

Output :
Y0 = Q0'Q1'X0' + Q0'Q1X0 + Q0Q1'X0 + Q0Q1X0'
Y1 = Q1

Flip-flop Input :

Use RS Flip-flop :          Use JK Flip-flop :
S0 = Q0'Q1'X0' +            J0 = Q1'X0' + Q1X0
Q0'Q1X0                     K0 = Q1'X0' + Q1X0
R0 = Q0Q1'X0' +             J1 = 1
Q0Q1X0                      K1 = 1
S1 = Q1'
R1 = Q1                     Use D Flip-flop :
                            D0 = Q0'Q1'X0' +
Use T Flip-flop :           Q0'Q1X0
T0 = Q1'X0' + Q1X0          D1 = Q1'
T1 = 1
```

Gambar 17. Uji Coba Tambahan Untuk *Up-down Counter*

8. ANALISA DAN KESIMPULAN

Pada uji coba rangkaian kombinasional, semua model untuk rangkaian kombinasional yang diuji, dapat dibentuk dengan hanya sebuah *contoh input/output sequence.* Hal ini karena sistem menganggap contoh-contoh yang diberikan cukup lengkap untuk membentuk sebuah model. Jika contoh yang diberikan tidak lengkap, maka perangkat lunak ini akan memberikan nilai *don't care* untuk informasi-informasi yang tidak jelas, dan model tetap dapat dibentuk, meskipun ternyata model yang dihasilkan tidak sesuai dengan keinginan. Karena itu, untuk rangkaian kombinasional, contoh yang paling lengkap, adalah sama dengan tabel kebenaran dari rangkaian logika yang dinginkan, yang memuat seluruh kemungkinan kombinasi input.

Pada uji coba rangkaian sekuensial, ternyata terdapat model yang tidak dapat dibentuk dengan

baik. Pada uji coba *up-down counter*, perangkat lunak menghasilkan sebuah model yang kemudian diberi contoh tambahan. Ternyata model yang dihasilkan pertama kali tidak konsisten dengan contoh yang diberikan, dan sistem ini mencoba membentuk model baru. Model yang baru dibentuk, ternyata juga konsisten dengan contoh sebelumnya, sehingga *weight*nya bertambah 1. Dari dua model yang dihasilkan, model kedua ternyata lebih sesuai dengan rangkaian logika yang dinginkan, karena memiliki *weight* paling besar. Kesalahan yang terjadi pada pembentukan model pertama kali adalah karena contoh yang diberikan pertama kali tidak lengkap. Jadi, keberhasilan perangkat lunak dalam membangun sebuah model, pada dasarnya tergantung atas informasi yang diberikan perancang dalam bentuk contoh *input/output sequence*.

Dari uji coba yang dilakukan diperoleh kesimpulan bahwa :

- Perangkat lunak mampu menghasilkan paling tidak sebuah model untuk rangkaian logika, berdasarkan atas contoh input/*output sequence* yang diberikan.
- Perangkat lunak mampu membedakan jenis dari rangkaian logika yang sedang dirancang.
- Perangkat lunak mampu menghasilkan model untuk rangkaian kombinasional cukup dengan sebuah contoh input/*output sequence* saja.
- Perangkat lunak mampu melakukan modifikasi atas model-model yang dihasilkan, berdasarkan atas informasi-informasi yang diperoleh dari contoh-contoh input/*output sequence* tambahan yang diberikan
- Perangkat lunak mampu mengidentifikasi inkonsistensi informasi yang diperoleh, dan mampu menangani inkonsistensi informasi tersebut.
- Tingkat kesesuaian model yang dihasilkan dengan model yang dinginkan adalah bergantung pada kelengkapan informasi yang diberikan melalui contoh *input/output sequence*.

Kemungkinan untuk pengembangan selanjutnya dari perangkat lunak ini adalah sebagai berikut :

(1) Sistem ini saat ini terbukti mampu menghasilkan model untuk rangkaian-rangkaian logika yang sederhana. Hal ini perlu dikembangkan agar perangkat lunak ini mampu menghasilkan model untuk rangkaian logika yang kompleks.

(2) Sistem ini saat ini hanya mampu menerjemahkan model yang dihasilkan ke dalam fungsi logika dalam bentuk *sum of products*. Kemampuan ini perlu dikembangkan hingga perangkat lunak mampu menghasilkan fungsi logika dalam bentuk-bentuk lainnya, baik yang standar atau yang tidak, serta mampu menerjemahkan model ke dalam bahasa perangkat keras seperti *VHDL*.

(3) Menambahkan algoritma untuk pengkodean state dalam FSM, sehingga diperoleh pengkodean yang optimal dan menghasilkan fungsi logika yang optimal juga.

(4) Menambahkan kemampuan untuk membangun *layout* yang baik, untuk FSM dalam bentuk *transition graph*, serta layout untuk fungsi logika dalam bentuk rangkaian gerbang-gerbang logika agar output dari perangkat lunak ini mudah dibaca oleh penggunanya.

9. DAFTAR PUSTAKA

[1] Choi B, *Applying learning by example for digital design automation*. Applied Intelligence Vol. 16. Kluwer Academic Publishers. 2002.

[2] Pena JM, Olievera AL. *A new algorithm for the reduction of incompletely specified finite state machines*. International Conference on Computer Aided Design. 1998.

[3] Kam T, Villa T, Brayton RK, Sangiovanni-Vincentelli A. *A fully implicit algorithm for exact state minimization*. 31st ACM/IEEE Design Automation Conference. 1994.

[4] Sanchez JM, Garnica AO. *A genetic algorithm for reducing the number of states in finite state machines*. Micro Electronics Journal Vol. 26. 1995.

[5] Fuhrer RM, Norwick SM. *OPTIMIST : State minimization for optimal 2-level logic implementation*. In : IEEE International Conference on Computer-Aided Design (ICADD-97). 1997.

[6] Ahmad I, Das AS, *A heuristic algorithm for minimization of incompletely specified finite state machines*. Computer And Electrical Engineering Journal Vol. 27. 2001.

[7] Hoguchi H, Matsunaga Y. *A fast state reduction algorithm for incompletely specified finite state machines*. ACM/IEEE 33rd Design Automation Conference, 1996.

[8] Hoguchi H, Matsunaga Y, *Implicit prime compatible generation for minimizing incompletely specified finite state machines*.

[9] Qu G. *Sequential system systhesis – incompletely specified machines*. http://www.ece.umd.edu/class/ enee644/

[10] Mano M. *Digital design*. Prentice Hall International Inc.

[11] Belton D. *Minimization of boolean functions, combinational logic and system tutorial guide*. University of Surrey UK. 1998.