

Pengantar Metoda Formal

Budi Rahardjo

5 Mei 2005

Daftar Isi

1	Pendahuluan	5
1.1	Latar Belakang	6
1.2	Kecelakaan akibat software atau hardware	8
1.3	Proses Desain	11
1.4	Prasyarat Pengetahuan	13
2	Formal Specification	15
2.1	Notasi formal	16
2.2	CSP	17
2.2.1	Konsep dan Notasi	17
2.2.2	Input dan Output	18
2.3	PROMELA dan SPIN	20
2.3.1	Bahasa PROMELA	20
2.3.2	Tool SPIN	23
2.4	Higher Order Logic	24
2.5	Petri nets	24
2.6	Synchronized Transition	25
2.7	Lain-lain: CCS, Estelle, Lotos, VDM, Z	25
3	Formal Synthesis	27
4	Formal Verification	29
4.1	Equivalence Checking	30
4.2	Model Checking	30
4.3	Theorem Proving	30
4.4	Contoh masalah	31
4.4.1	Truth table	31
4.4.2	Karnaugh-map	32
4.4.3	Aljabar Boolean	32

4.4.4	Propositional Tableaux	33
4.5	Binary Decision Diagram	33
4.5.1	Binary Decision Tree	33
4.5.2	Dari Binary Decision Tree ke BDD	35
4.5.3	OBDD	40
4.5.4	Operasi terhadap OBDD	41
4.5.5	Tools	43
4.6	Verification Tools	43
5	Studi Kasus	45
A	Laboratorium BDD	47
A.1	Masuk ke sistem UNIX	47
A.2	Mencoba BDD	48
A.3	Equivalency, isomorphism	49
A.4	Efek dari ordering	50
A.5	Lain-lain	51

Informasi mengenai buku ini.

Hak Cipta © 2004,2005 Budi Rahardjo. Buku ini memiliki lisensi Creative Commons¹. Buku ini dapat digunakan untuk keperluan apa saja dan didistribusikan secara gratis.

¹Informasi mengenai Creative Commons dapat dilihat pada situs <http://creativecommons.org>

Bab 1

Pendahuluan

Buku ini merupakan panduan kuliah Metoda Formal (dalam desain perangkat keras) atau *Formal Methods for Hardware Design* yang diberikan oleh Departemen (Jurusan) Teknik Elektro ITB. Buku ini merupakan kelanjutan dari *technical report* [12] yang merupakan hasil dari Riset Unggulan Terpadu (RUT) dan rangkuman dari beberapa makalah yang relevan dengan bidang metoda formal ini.¹

Beberapa bagian dari buku ini membutuhkan latar belakang matematika diskrit dan *logic*. Agar bisa berdiri sendiri, buku ini memberikan beberapa pengantar tentang matematika diskrit dan *logic*. Namun tentunya buku ini bukan buku utama untuk hal itu. Bagi anda yang tertarik dengan referensi yang lebih komplit silahkan gunakan buku Kenneth Rosen [13].

Demikian pula beberapa topik, seperti notasi formal yang ada pada bab Formal Specification, masing-masing dapat menjadi buku tersendiri. Memang pada kenyataannya banyak buku untuk masing-masing topik tersebut. Namun buku ini bermaksud mengambil bagian yang penting untuk sekedar memberikan wawasan kepada pembaca.

Buku ini berisi beberapa contoh yang diperoleh dari kelas yang telah diajarkan oleh penulis. Beberapa contoh menggunakan tools yang dapat diperoleh secara gratis dan komersial. Ada tools yang menggunakan sistem UNIX. Untuk bagian itu diharapkan pembaca dapat mempelajari UNIX sendiri.

¹Buku ini masih dalam proses penulisan sehingga masih banyak bagian yang belum selesai. Proses penulisan menggunakan L^AT_EX yang belum terkonfigurasi untuk bahasa Indonesia. Untuk itu anda akan menemukan beberapa kejanggalan pada pemenggalan kata.

Tabel 1.1: Perkembangan kompleksitas software

Sistem Operasi	Tahun	Jumlah Baris (lines of code)
Windows 3.1	1992	3 juta
Windows NT	1992	4 juta
Windows 95	1995	15 juta
Windows NT 4.0	1996	16,5 juta
Windows 98	1998	18 juta
Windows 2000	2002	35 s/d 60 juta

1.1 Latar Belakang

“There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.”

(C.A.R. Hoare)

Apa itu metoda formal? Mengapa metoda formal dibutuhkan? Apa hubungannya dengan desain hardware? Bagian ini akan mencoba menjawab pertanyaan tersebut. Metoda formal pada mulanya muncul di dunia software (perangkat lunak)². Namun istilah metoda formal ini juga muncul di dunia hardware dan bahkan kelihatannya penerapan metoda formal lebih banyak suksesnya di bidang hardware dibandingkan dengan bidang software.

Jika kita perhatikan tingkat kompleksitas dari desain software dan hardware, maka kita dapat melihat kenaikan yang cukup tajam. Bahkan *trendnya* menunjukkan tanda-tanda makin meningkat. Sebagai contoh, software *word-processor Wordstar* (WS) dahulu cukup dimasukkan ke dalam disket dalam ukuran 300 KBytes. Sementara ini, program *Microsoft Word 2000* harus didistribusikan dalam CD-ROM yang ukurannya ratusan MBytes. Sebagai gambaran, tabel 1.1 menunjukkan perkembangan kompleksitas dari operating system Microsoft Windows. Kompleksitas dalam tabel tersebut ditunjukkan dengan banyaknya jumlah baris kode (*lines of code (LOC)*). Sebagai perkiraan, setiap KLOC dapat memiliki 5 sampai dengan 50 bugs.

Jumlah transistor dalam sebuah *Integrated Circuit* (IC) sudah mencapai jutaan transistor. Pentium chip memiliki jutaan transistor. Kesemuannya

²Banyak istilah asing yang dipaksakan diterjemahkan ke dalam bahasa Indonesia. Hasilnya justru malah membingungkan. Istilah *software* akan digunakan dalam buku ini agar tidak membingungkan.

Tabel 1.2: Kompleksitas software lainnya [7]

Sistem	Jumlah Baris Kode
Solaris 7	400.000
Linux	1,5 juta
Boeing 777	7 juta
Space Shuttle	10 juta
Netscape	17 juta
Space Station	40 juta
Windows XP	40 juta

ini membutuhkan konsep desain yang berbeda dengan mendesain software atau hardware yang ukurannya kecil.

Jika kompleksitas ini belum terbayang, maka anda dapat mencoba membayangkan tingkat kesulitan dalam mendesain rumah 1 tingkat dengan mendesain hotel 150 tingkat. Desain rumah 1 tingkat dapat anda percayakan kepada seorang sarjana yang baru saja lulus dari perguruan tinggi. Tapi, apakah anda akan mempercayakan desain hotel 150 tingkat kepada seorang mahasiswa yang baru lulus? Kemungkinan besar, tidak. Demikian pula, apakah metodologi desainnya akan sama? Mendesain rumah 1 tingkat bisa dilakukan dengan “coba-coba”, tapi mendesain hotel 150 tingkat tidak dapat dilakukan dengan cara coba-coba.

Paradigma atau metoda desain yang baru dibutuhkan untuk menangani tingkat kompleksitas yang tinggi, khususnya untuk mengurangi jumlah kesalahan (*bugs*) dan mempercepat waktu yang dibutuhkan untuk mengembangkan dan memasarkan. *Time to market* di dunia yang *high-tech* ini sangat sempit. Khususnya untuk aplikasi yang membutuhkan keandalan tinggi (yang sering disebut *mission critical applications*), perhatian dalam desain dan pengujian perlu mendapat perhatian yang lebih. Aplikasi yang membutuhkan ketelitian tinggi antara lain aplikasi untuk kesehatan (misalnya alat pacu jantung), sistem pengendali rudal nuklir atau pesawat ruang angkasa, pengendali pesawat terbang (*fly-by-wire*), dan masih banyak aplikasi sejenis lainnya.

Ada beberapa masalah dalam desain dan pengujian sistem yang membutuhkan tingkat keandalan tinggi. Metoda formal membantu di bidang ini dengan penggunaan metoda matematik yang lebih *rigorous*.

1.2 Kecelakaan akibat software atau hardware

Untuk meyakinkan bahwa masalah ini bukan hanya masalah akademis saja, pada bagian ini akan didaftar beberapa contoh kasus kecelakaan yang disebabkan oleh adanya kesalahan (*bugs*) pada software dan/atau hardware. Akibat yang ditimbulkan oleh kesalahan ini bervariasi dari ketidak-nyamanan (*inconvenience*) sampai korban jiwa (fatal).

1. **22 Juli 1962.** Roket pembawa Mariner I (yang akan digunakan untuk menjelajah ke Venus) terpaksa diledakkan. Penyebab utamanya adalah persamaan matematis yang digunakan untuk mengendalikan roket pembawa Mariner I kekurangan tanda “bar” (garis di atas simbol yang digunakan untuk menyatakan rata-rata atau average). Akibat kesalahan ini, komputer yang digunakan untuk mengendalikan roket menyatakan bahwa roket tak terkendali, meskipun sebetulnya roket tidak apa-apa. Akibatnya roket terpaksa harus dihancurkan untuk menghindari kecelakaan yang berakibat lebih fatal. Sumber [15].
2. **10 April 1981.** *The bug heard round the world.* Pesawat ulang alik Columbia batal diluncurkan. Penyebabnya adalah hardware. Pada tanggal ini, pesawat ulang alik (space shuttle) Columbia batal diluncurkan 20 menit sebelum waktu yang sudah direncanakan. Penyebabnya adalah tidak sinkronnya clock komputer utama dan komputer backup. Untuk meningkatkan keandalan, komputer pengendali dari pesawat ulang alik tersebut terdiri dari empat (4) komputer utama dan sebuah komputer backup. Ternyata kadang-kadang (pobabilitasnya 1 dari 67) clock komputer utama (primary) lebih cepat satu cycle dari komputer backup ketika dihidupkan. Akibatnya komputer backup menunggu dan menunggu untuk sebuah sinyal yang tidak kunjung datang (deadlock). Peluncuran terpaksa dibatalkan. Sumber: [10].
3. **25 Februari 1991.** Di bidang militer, Patriot Amerika gagal menanggulangi roket *scud* milik Irak. Pada perang antara Irak dan Amerika ini, roket *scud* dari Irak menewaskan 28 tentara Amerika dan menciderai 98 tentara di barak dekat Dhahran, Saudi Arabia. Penyebabnya adalah kesalahan software. *Patriot missile defense system* menggunakan software untuk melakukan *scanning* ke angkasa dengan menggunakan radar (yang memiliki lebih dari 5000 elemen) sampai dia menemukan targetnya. Data radar yang diterima menunjukkan kecepatan dari target tersebut. Sistem Patriot kemudian menentukan arahnya.

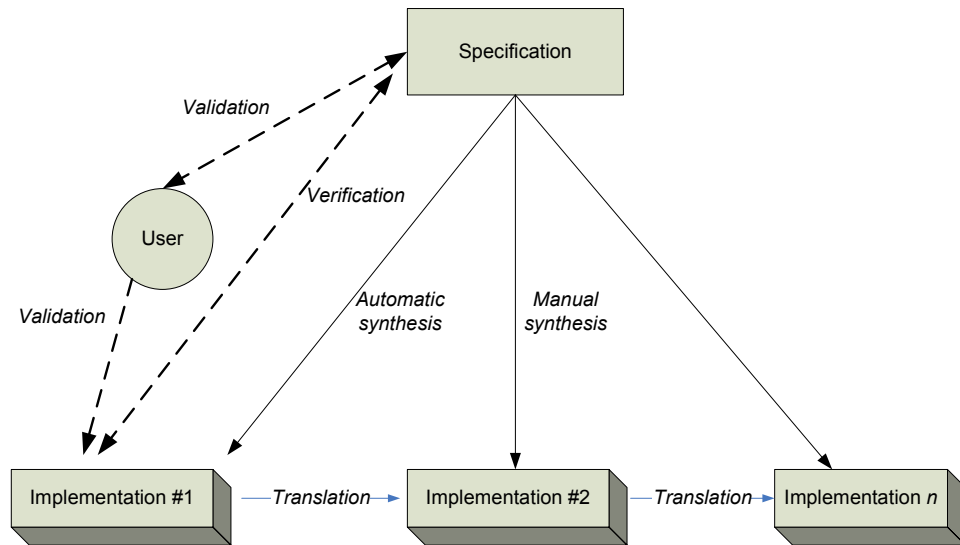
Persamaan yang digunakan untuk melakukan *tracking* memiliki kesalahan 1/10 juta detik dalam setiap detiknya. Kesalahan ini dapat terakumulasi dan menjadi besar. US Army membuat prosedur baku bahwa mesin harus direset secara berkala untuk menghilangkan akumulasi kesalahan tersebut. Diperkirakan sistem Patriot hanya dapat digunakan secara maksimal selama 14 jam berturut-turut. Pada waktu kejadian, sistem Patriot sudah digunakan selama 5 hari berturut-turut sehingga timingnya sudah bergeser sebesar 36/100 detik (sudah cukup besar). Kesalahan ini sebetulnya sudah diketahui, akan tetapi perbaikan (*upgrade*) yang dikerjakan di *Ft. McGuire Air Force base* membutuhkan waktu untuk mencapai tujuan dengan cara diterbangkan ke Riyadh, kemudian dikirimkan melalui truk ke Dhahran, dan akhirnya dipasang di tempat instalasi Patriot. Ternyata sudah terlambat. Sumber [15].

4. **4 Juni-Juli 1991.** Di bidang telekomunikasi, telepon di daerah Los Angeles, San Francisco, Washington DC, West Virginia, Baltimore, Greensboro terputus. Penyebabnya adalah kesalahan software. Software yang digunakan untuk mengatur *telephone switching* dibuat oleh perusahaan DSC Communications. Software ini terdiri atas jutaan baris. Program sudah diuji selama 13 minggu dan sudah berjalan dengan sempurna. Kemudian sebelum instalasi mereka “memperbaiki” 3 baris kode saja dan merasa tidak perlu diuji kembali (karena akan menghabiskan 13 minggu lagi). Akibat “perbaikan” ini sistem justru menjadi *crash*. Sumber [15].
5. **4 Juni 1996.** Roket Ariane 5 meledak 40 detik setelah diluncurkan. Sumber informasi: European Space Agency (www.esa.int).
6. Radar dari HMS Sheffield salah mengidentifikasi roket Exocet milik Argentina sebagai roket non-Soviet sehingga dianggap sebagai kawan. Tidak ada alarm yang dinyalakan. Akibatnya kapal tersebut tenggelam beserta awak kapalnya. Kelalaian ini disebabkan oleh *design errors*. Sumber informasi: ACM Software Engineering Notes 8 (3).
7. **Juni 2004.** Air traffic controller di Inggris (West Drayton control center) bermasalah (down) karena adanya “upgrade”. Penerbangan terpaksa diundurkan (delay) sehingga menyebabkan ketidaknyamanan penumpang. Tidak ada korban dan tidak ada informasi yang lebih lanjut mengenai masalah “update” tersebut.

Sumber berita: <http://news.bbc.co.uk/1/hi/uk/3772663.stm>
http://www.nats.co.uk/news/news_stories/2004_06_03.html

8. *Tanggal tidak diketahui.* NASA Mars Lander menabrak permukaan Mars. Kesalahan terjadi karena ada perbedaan satuan ukuran yang digunakan, yaitu antara Inggris dan metric unit. Akibatnya, trayektori dari Mars Lander tersebut menjadi salah. Sang Lander mematikan mesinnya terlalu dini sehingga terjadi crash. Kerugian diperkirakan 165 juta dolar.
9. *Tanggal tidak diketahui.* Denver International Airport mengimplementasikan sistem penanganan bagasi secara otomatis, dengan menggunakan jalur-jalur (track) yang semuanya dikendalikan dengan software. Pada saat pengujian ternyata kereta bagasi tidak dapat mendeteksi kesalahan dan tidak dapat kembali (recover) dari error. Kereta yang kosong tidak diisi. Sementara itu kereta yang sudah berisi bagasi diisi lagi sehingga melebihi beban. Penyebabnya juga software. Akhirnya pengoperasian sistem ini terlambat 11 bulan dan mengakibatkan kerugian 1 juta dolar seharusnya.
10. *Tanggal tidak diketahui.* MV-22 Osprey merupakan sebuah pesawat militer yang menggabungkan helikopter (yang dapat bergerak vertikal) dan pesawat terbang biasa. Tingkat kompleksitas dari aerodinamik dari pesawat ini sangat kompleks sehingga dibutuhkan software canggih untuk mengendalikannya. Pesawat ini juga menggunakan sistem redundan untuk meningkatkan keandalannya. Namun pada suatu saat terjadi kerusakan pada sistem hydraulic. Ini memang masalah serius, namun biasanya bisa dikendalikan. Sanganya pada saat masalah ini terjadi, sistem backup tidak berjalan sebagaimana mestinya. Pesawat jatuh dan empat orang marine mati.
11. **1988. US Vicennes.** Di tahun 1988, sebuah kapal laut Amerika menembakkan peluru kendali dan menjatuhkan sebuah pesawat yang diidentifikasi sebagai musuh. Ternyata pesawat yang ditembak adalah sebuah pesawat komersial Airbus A320 yang sangat jauh berbeda dengan pesawat musuh. Akibatnya 290 penumpang pesawat tersebut tewas. Angkatan laut Amerika menyalahkan sistem penjejak (*tracking software*) yang memperagakan output yang tidak dapat dimengerti (*cryptic*) sehingga mengambil kesimpulan yang salah.

Selain kasus-kasus di atas, sebetulnya masih banyak kecelakaan lain yang terjadi di dunia *engineering* yang tidak terkait dengan hardware atau soft-



Gambar 1.1: Diagram proses desain

ware. Buku Henry Petroski [11] menceritakan beberapa kesalahan dalam desain jembatan dan bangunan yang berakibat fatal. European Space Agency (ESA) memiliki situs web (www.esa.int) yang memberitakan berbagai masalah tentang penerbangan ruang angkasa. Usenet newsgroup *comp.risk* juga banyak menceritakan kecelakaan yang diakibatkan oleh salah desain.

1.3 Proses Desain

Secara umum, proses desain meliputi beberapa tahap; spesifikasi (*specification*), implementasi (*implementation*), dan validasi (*validation*) serta verifikasi (*verification*). Kegiatan lain yang terkait dengan tahapan tersebut antara lain adalah *automated synthesis* dan *translation*. Kerangka tahapan tersebut dapat dilihat pada gambar 1.1.

Proses desain dimulai dari sebuah spesifikasi (*specification*) dari sistem (software atau hardware) yang akan dibuat. Spesifikasi ini dibuat oleh pemberi pekerjaan, yang bisa sama dengan orang yang akan melakukan desain tetapi tidak harus sama. Spesifikasi ini kemudian akan diimplementasikan. Masalah yang timbul adalah bagaimana cara menuliskan spesifikasi agar dapat dimengerti oleh orang yang akan mengimplementasikan spesifikasi tersebut. Spesifikasi yang ditulis dalam bahasa natural (*natural language*), seperti bahasa Indonesia atau bahasa Inggris, sering membingungkan kare-

na memiliki kerancuan makna (*ambiguous*). Untuk itu perlu dibuat sebuah standar formal untuk mendefinisikan spesifikasi software atau hardware. Pembahasan mengenai *formal specification* ini akan dibahas pada bab terpisah.

Sebuah spesifikasi dapat diterjemahkan menjadi beberapa implementasi, *n-implementations*. Ada cara melakukan implementasi secara otomatis melalui *automatic or automated synthesis* atau implementasi secara manual³. (Lihat gambar 1.1.) Sebuah implementasi dapat *ditranslasikan* menjadi implementasi lain melalui proses optimasi. Sebagai contoh, di dalam desain hardware kita dapat melakukan optimasi untuk menghemat area atau menghemat power. Hasil dari proses optimasi ini adalah implementasi lain yang sudah *optimized*. Pengertian translasi adalah perubahan pada satu level desain yang sama.

Setelah implementasi selesai, maka tugas seorang desainer adalah menguji dan menjamin kebenaran dari implementasinya. Proses pengujian ini biasanya dilakukan dengan simulasi atau testing secara coba-coba atau *ad-hoc*. Testing dilakukan dengan mengambil sampel atau menguji dengan semua kombinasi (*exhaustive test*). Hal ini hanya dapat dilakukan untuk sistem yang berukuran kecil. Sebagai contoh, untuk sistem hardware digital yang memiliki empat (4) input, ada 2^4 atau 16 kombinasi yang harus dicoba. Angka ini cukup kecil untuk *exhaustive test*. Untuk sistem yang memiliki ukuran besar, misalnya mikroprosesor yang memiliki data 64-bit, *exhaustive test* tidak dapat dilakukan. Kombinasi 2^{64} sudah sangat besar sekali. Dapat dibayangkan apabila kita harus menguji sistem yang memiliki 128-bit input.

Apabila anda belum dapat membayangkan betapa besarnya angka tersebut, lihat tabel 1.3. Dapat dilihat bahwa kemungkinan anda mendapatkan lotre dan disambar petir lebih besar daripada menemukan bugs pada sebuah sistem yang memiliki jumlah bit 64-bit.

Proses pengujian dengan menggunakan simulasi dan sampel dari data disebut sebagai validasi (*validation*). Sementara itu proses verifikasi (*verification*) adalah membuktikan secara matematis bahwa implementasi memang mengimplementasikan spesifikasi. Verifikasi secara formal menjamin bahwa implementasi memang mengimplementasikan spesifikasi. Namun masih ada kemungkinan bahwa spesifikasinya itu sendiri yang salah, yaitu bukan yang diinginkan oleh desainer. Karena spesifikasinya bukan yang diinginkan, maka implementasinya akan salah karena dia mengikuti spesifikasi tersebut, meskipun implementasi sudah diverifikasi. Jadi validasi masih dibutuhkan untuk memastikan bahwa implementasi memang yang diinginkan. Formal

³Pada versi berikutnya akan diceritakan mengenai formal synthesis.

Tabel 1.3: Large numbers [14]

Besaran fisik	Besaran angka
Probabilitas terbunuh karena sambaran petir setiap harinya	1 in 9 billion (2^{33})
Probabilitas memenangkan hadiah pertama dari lotre di Amerika Serikat	1 in 2^{28}
Probabilitas memenangkan hadiah pertama dari lotre di Amerika Serikat dan terbunuh oleh sambaran petir di hari itu juga	1 in 2^{61}
Umur dari sebuah planet	2^{30}
Umur dari universe	2^{34}
Jumlah atom di planet	2^{170}

verification akan dibahas pada bab tersendiri.

1.4 Prasyarat Pengetahuan

Untuk mempelajari metoda formal secara baik dibutuhkan beberapa pengetahuan dasar seperti matematika, Finite Automata (FA) atau Finite State Machine (FSM), dan teori bahasa (seperti penggunaan notasi BNF, *parser*). Setiap pengetahuan tersebut di atas memiliki buku atau referensi yang lebih baik dari buku ini. Sayangnya buku-buku yang membahas hal tersebut biasanya ukurannya sangat tebal dan isinya agak “berat”. Contohnya:

- Buku karangan K. Rosen untuk Discrete Math
- Hopcroft dan Ullman

Apabila memungkinkan, hal-hal yang dibutuhkan untuk memahami sebuah konsep yang ada di dalam buku ini akan dijelaskan dengan cara membuat sari atau rangkuman dari buku-buku tersebut di atas.

Bab 2

Formal Specification

Seperti telah dijelaskan secara sepintas pada bab Pendahuluan, proses desain dimulai dari sebuah spesifikasi. Spesifikasi ini dibuat oleh pemberi pekerjaan yang kemudian akan diimplementasikan oleh seorang (atau satu tim) implementor.

Spesifikasi dari sebuah sistem yang akan didesain dapat dituliskan dalam bahasa Inggris (atau *natural language* lainnya) dan kadang-kadang dilengkapi dengan diagram atau sketsa. Namun spesifikasi dalam bahasa natural ini memiliki kelemahan:

- *ambiguous* atau memiliki banyak makna
- sukar diproses secara otomatis (*mechanized*)

Contoh suatu pernyataan atau spesifikasi yang *ambiguous* adalah sebagai berikut. Jika kita pergi ke sebuah restoran, ada sebuah pilihan:

Sup atau sayur sudah termasuk dalam menu.

Apakah yang dimaksud adalah salah satu saja (sup saja, atau sayur saja)? Ataukah keduanya? Kata “atau” di sini sangat rancu. Dia mungkin tidak tepat sama seperti *OR* dalam boolean logic dimana keduanya boleh ada, akan tetapi lebih tepat seperti *exclusive OR* atau *XOR* dimana hanya salah satu yang dipilih tapi tidak boleh dua-duanya.

Spesifikasi yang dituliskan dalam bahasa natural tadi, karena kerancuannya, sulit untuk diproses secara otomatis oleh sebuah komputer misalnya. Jika kita dapat menggunakan notasi yang standar dalam penulisan spesifikasi, mungkin proses selanjutnya bisa diotomatisasi dengan lebih mudah. Sebagai contoh, Kimia mempunyai standar baku untuk menuliskan rumus-rumus Kimia.

TODO: contoh-contoh notasi kimia

Contoh lain yang dapat menunjukkan kemudahan notasi formal adalah penulisan rumus matematik. Sebagai contoh, *Fermat's Last Theorem* dapat dijabarkan dalam kata-kata sebagai berikut:

Tidak ada empat bilangan bulat yang memenuhi persamaan sebagai berikut. Bilangan pertama dipangkatkan bilangan keempat, ditambah bilangan kedua dipangkatkan bilangan keempat, sama dengan bilangan ketiga dipangkatkan bilangan keempat, dimana bilangan keempat lebih besar dari dua.

Teorema di atas lebih mudah dituliskan sebagai berikut

Tidak ada bilangan bulat x, y, z yang memenuhi persamaan $x^n + y^n = z^n$, dimana $n > 2$.

Untuk mengatasi masalah spesifikasi yang tidak jelas maka dibuatlah pendekatan yang disebut *formal specification*. Hal ini dilakukan dengan menggunakan notasi formal.

2.1 Notasi formal

Formal specification menggunakan notasi formal yang memiliki basis matematik serta memiliki *syntax* dan *semantics* yang jelas. Seringkali notasi formal ini memiliki format yang dapat diproses oleh komputer (*machine readable format*). Tujuannya adalah membuat proses verifikasi menjadi otomatis, yaitu melakukan perbandingan antara spesifikasi dengan implementasi secara otomatis oleh komputer.

Pada bagian ini akan dibahas beberapa contoh notasi formal. Pemilihan notasi bergantung kepada beberapa hal:

- expresiveness
- kemudahan penggunaan (easy to use)

Kemampuan ekspresi yang tinggi memudahkan untuk menulis spesifikasi sistem yang kompleks. Namun seringkali hal ini diimbangi dengan ketidakmudahan untuk menuliskan spesifikasi tersebut. Sebagai contoh, higher-order logic memiliki kemampuan ekspresi yang sangat fleksibel. Namun untuk menggunakan higher-order logic ini membutuhkan pengetahuan dan pengalaman yang tinggi (*steep learning curve*). Kesulitan ini memudahkan orang membuat kesalahan dalam menuliskan spesifikasi, atau spesifikasi itu susah dibaca dan dimengerti.

2.2 CSP

Communicating Sequential Processes (CSP) merupakan sebuah *formalism* yang dikembangkan oleh C.A.R. Hoare [5, 6]. CSP ini memiliki kemampuan ekspresi yang sangat baik. Namun sayangnya belum ada tools CSP yang mudah digunakan dan tersedia secara gratis. (Lihat bagian PROMELA.)

Pada bahasa pemrograman tingkat tinggi, *assignment* merupakan sebuah hal yang sudah dimengerti dengan baik. Akan tetapi operasi *input* dan *output* belum dimengerti dengan baik. *Assignment* berhubungan dengan *state* di dalam (internal) sebuah sistem, sementara *input* dan *output* berhubungan dengan *state* di luar (external). CSP mencoba memberikan sebuah notasi untuk input dan output ini.

Program konvensional ditargetkan untuk dijalankan pada satu prosesor. Saat ini mulai banyak sistem yang menggunakan prosesor lebih dari satu (multiprocessor), dimana prosesor ini saling berkomunikasi satu dengan lainnya. Komunikasi antar prosesor ini harus dapat dijabarkan dengan mudah dan tidak *ambiguous*. Itulah sebabnya perlu ada notasi input dan output.

Communicating Sequential Processes (CSP) mencoba memecahkan masalah notasi untuk menjabarkan (menspesifikasi) sistem yang terdiri dari beberapa proses, dimana pada setiap proses ini ada program yang sifatnya berurutan (sequential). Proses-proses ini saling berkomunikasi satu dengan lainnya melalui input dan output. Contoh penggunaan notasi ini akan dijelaskan lebih lanjut.

CSP mengusulkan sebuah solusi, yaitu:

- *Dijkstra's guarded command*;
- Perintah parallel, yang juga berdasarkan kepada *Dijkstra's* parbegin;
- Perintah input dan output dengan format yang sederhana;
- dan gabungan hal di atas.

2.2.1 Konsep dan Notasi

Notasi dari CSP bisa dituliskan dengan menggunakan notasi BNF (Backus-Naur form)¹.

¹Backus-Naur form, yang juga dikenal dengan istilah Backus normal form, merupakan sebuah metasyntax untuk menjelaskan context-free grammars. Istilah BNF diambil dari dua nama, yaitu John Backus dan Peter Naur. BNF banyak digunakan sebagai notasi untuk mendeskripsikan grammar dari bahasa pemrograman. BNF pertama kali digunakan untuk mendeskripsikan sintaks dari bahasa ALGOL 60. Sejarahnya, Backus pertama kali

```

<command> ::= <simple command>|<structured command>
<simple command> ::= <null command>|<assignment command>
    |<input command>|<output command>
<structured command> ::= <alternative command>
    |<repetitive command>|<parallel command>
<null command> ::= skip
<command list> ::= {<declaration>;|<command>;} <command>

```

Sebuah *command* (perintah) menspesifikasikan sifat dari sebuah perangkat (sistem) yang mengeksekusi perintah tersebut. Eksekusi dari sebuah *simple command*, jika berhasil, bisa menghasilkan efek terhadap state internal dari device tersebut (jika ini berupa sebuah *assignment*, atau terhadap lingkungan eksternal (jika itu berupa perintah *output*, atau keduanya (jika itu berupa perintah *input*).

Eksekusi dari *null command* selalu berhasil dan tidak memiliki efek.

2.2.2 Input dan Output

Perintah atau operator input dan output digunakan untuk menspesifikasikan komunikasi antara dua proses yang berjalan bersamaan (*concurrent*). Jika kita memiliki dua proses, P1 dan P2, yang berjalan bersamaan dan saling berkomunikasi maka komunikasinya dapat dilakukan dengan:

- Perintah input di proses P1 yang menspesifikasikan proses P2 sebagai sumbernya (*source*). Perintah input dituliskan dengan operator tanda tanya (?).
- Perintah output di proses P2 yang menspesifikasikan proses P1 sebagai tujuannya (*destination*). Perintah output dituliskan dengan operator tanda seru (!).
- Variabel target cocok dengan nilai (*value*) yang dituliskan pada ekspresi perintah output.
- Perintah input berhenti (gagal, *fail*) apabila sumber data dihentikan (*terminated*).
- Perintah output akan gagal bila proses target berhenti (terminated).

menggunakan sebagian dari notasi BNF untuk menjelaskan bahasa ALGOL 58. Peter Naur membaca laporan Backus dan terkejut karena interpretasi dia terhadap ALGOL 58 berbeda dengan interpretasi Backus. Untuk selanjutnya disepakati bahwa desain dari bahasa ALGOL harus dituliskan dengan notasi BNF tersebut.

Beberapa contoh penggunaan notasi dari CSP antara lain:

- **cardreader?cardimage**
 baca data dari proses “cardreader” dan masukkan ke variabel “cardimage”
- **lineprinter!lineimage**
 kirimkan data dari variabel “lineimage” ke proses “lineprinter”
- **[cardreader?cardimage] || [lineprinter!lineimage]**
 Notasi di atas menunjukkan dua perintah secara paralel dan berhenti apabila kedua operasi di atas telah selesai.
- **X?(x,y)**
 Baca dari proses X dua data yang akan dimasukkan ke variabel x dan y . Data yang dikirimkan harus sama jenisnya (tipenya) dengan kedua variabel tersebut. Jika tipenya berbeda, maka proses input tersebut tidak bisa dieksekusi.
- **DIV!(3*a+b,13)**
 Kirim ke proses DIV dua buah data, dimana data yang pertama adalah nilai hasil evaluasi $3 * a + b$ sementara nilai kedua adalah 13.
- Kedua perintah di atas dapat digabungkan menjadi satu sehingga berarti $(x, y) = (3 * a + b, 13)$, yaitu masukkan hasil evaluasi $3 * a + b$ ke variabel x dan angka 13 ke variabel y .
- **console(i)?c**
 Baca data dari *console* nomor i , dimana angka i merupakan integer.
- **console(j-1) ! “A”**
- **X(i)?V()**
 Baca dari proses $X(i)$ sebuah sinyal $V()$. Sinyal ditandai dengan tanda kurung. Jika sinyal tersebut tidak bisa dibaca, misalnya proses $X(i)$ mengirimkan sinyal lain, maka ekspresi tersebut tidak dapat dieksekusi.
- **sem!P()**

(Pada versi berikutnya akan saya tambahkan informasi mengenai notasi CSP di sini.) Penjelasan mengenai notasi CSP dibahas dengan lebih lengkap pada paper Hoare [5].

2.3 PROMELA dan SPIN

PROMELA merupakan sebuah bahasa yang dikembangkan oleh Gerrard Holzmann [8] untuk memodelkan sistem, khususnya protokol komputer. PROMELA, yang merupakan kependekan dari PROcess MEta LAnguage, menggunakan basis CSP, bahasa C, dan SDL (*Specification Description Language*, yang akan dibahas tersendiri pada bagian terpisah).

Selain mengembangkan bahasanya, Holzmann juga mengembangkan sebuah tools yang diberi nama SPIN untuk melakukan validasi – tepatnya *model checking* – terhadap spesifikasi yang ditulis dalam bahasa PROMELA. Bahkan, sebetulnya buku dan makalah dari Holzmann lebih banyak difokuskan kepada algoritma-algoritma yang dituangkan dalam bentuk tools SPIN tersebut.

2.3.1 Bahasa PROMELA

Pada bagian berikut akan dijelaskan secara sepiantas mengenai bahasa PROMELA². Informasi yang lebih lengkap dapat dilihat pada buku referensi.

Secara singkat, di dalam bahasa PROMELA ada tiga buah *objects*;

- *process*
- *message channels*
- variabel (global atau lokal)

Dalam implementasinya, obyek ini ditranslasikan menjadi *Finite State Machine* (FSM).

Deskripsi dalam bahasa PROMELA ditulis dalam bentuk *statement*. Dalam PROMELA tidak ada perbedaan antara *conditions* dan *statements*. Eksekusi dari *statement* ini bergantung kepada sifat *executability*-nya. *Statement* ini bisa dijalankan (*executable*) atau terhalang (*blocked*) bergantung kepada nilai dari variabel atau isi dari message channel. Jika kondisinya benar (*holds*) maka *statement* akan dieksekusi. Jika tidak, maka *statement* akan terhenti (*blocked*) sampai kondisi menjadi benar. Jika keterangan di atas agak membingungkan, jangan khawatir. Pada bagian berikut kita akan melihat contoh-contoh.

Ada enam (6) jenis data yang sudah terdefinisi di PROMELA;

- *bit* (ukuran implementasi 1 bit)

²Informasi mengenai PROMELA dan SPIN dapat diperoleh dari situs <http://spinroot.com>

- *bool* (1 bit)
- *byte* (8 bit)
- *short* (16 bit)
- *chan* (bergantung kepada data). Jenis data *chan* ini digunakan untuk mengimplementasikan message channel, yaitu sebuah obyek yang dapat menyimpan beberapa nilai yang dijadikan satu.

Contoh penggunaan jenis data di atas:

```
bool flag;
int state;
byte pesan;
```

Variabel data dinyatakan dalam *array* seperti contoh di bawah ini:

```
byte status[N];
status[0] = status[2] + 7 * status[3*n]
```

Perlu dicatat bahwa *declaration* dan *assignment* selalu dapat dieksekusi, jadi selalu *executable*. Tanda titik koma (; atau *semi-colon*) merupakan tanda pemisah statement atau *statement separator*. Dia bukan *statement terminator* sehingga pada statement yang terakhir tidak perlu diakhiri dengan tanda titik koma ini. Ada dua *statement separator* dalam PROMELA, yaitu titik koma dan tanda panah $->$.

Proses di dalam PROMELA didefinisikan dengan menggunakan keyword *proctype*, seperti contoh di bawah ini.

```
proctype A() { byte state; state = 3}
```

Contoh di atas menunjukkan definisi dari proses A, dimana di dalamnya ada sebuah variabel lokal yang bernama *state*. Kemudian variabel ini kita masukkan nilai “3”.

Contoh lain lagi:

```
byte state = 2;
proctype A() { (state == 1) -> state = 3}
proctype B() { state = state 1}
```

Pada contoh ini kita melihat adanya sebuah variabel global yang bernama *state* dan pendefinisian dua buah proses, yaitu proses “A” dan “B”. Perhatikan bahwa pada proses A terdapat sebuah *guarded command* dimana eksekusi dari statement tersebut bergantung kepada nilai dari variabel *state*.

Contoh di atas hanya menunjukkan deskripsi dari proses tetapi belum menunjukkan interaksi antar proses tersebut selain melalui global variabel. Proses diinisialisasi dengan menggunakan perintah *run* seperti contoh di bawah ini:

```
init { run A(); run B() }
```

Perintah *init* ini sama seperti perintah *main()* pada bahasa C.

Message channel digunakan untuk mengkomunikasikan data antar proses. Di bawah ini ada beberapa contoh penggunaan message channel.

```
chan a, b; chan c[3];
chan qname = [16] of { byte, int, chan, byte};
qname!expr;
qname?msg
```

Mirip dengan CSP, nilai dari sebuah variabel dapat dikirimkan melalui channel dengan menggunakan tanda seru. Dalam contoh di atas, nilai dari ekspresi *expr* akan dikirimkan melalui channel “qname”. Sementara itu baris terakhir menunjukkan bahwa sebuah pesan dari kepala “qname” dibaca dan disipan dalam variabel “msg”. Channel bersifat FIFO (First In, First Out).

PROMELA memiliki tiga *control flow*; *case selection*, *repetition*, dan *unconditional jump*.

Pemilihan atau *case selection* dilakukan dengan menggunakan kata kunci *if* dan tanda titik dua (:) dua kali.

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

Dalam contoh di atas, statement pertama berfungsi sebagai guard, yang ikut menentukan sifat *executable* dari statement tersebut. Dalam satu saat dipilih salah satu dari pilihan tersebut dengan syarat bahwa statement tersebut executable. Dalam contoh di atas, hanya salah satu statement akan dipilih bergantung kepada guard yang ada, yaitu bergantung kepada nilai variabel “a” dan “b”. Namun dalam contoh lain bisa terjadi bahwa ada lebih dari satu pilihan yang bisa dipilih. Untuk kasus ini akan dipilih salah satu secara random.

```
if
:: (x > y) -> maks = x
```

```

:: (y > x) -> maks = y
:: (x == y) -> maks = x
:: (x != y) -> flag = 1
fi

```

Dalam contoh di atas jika $x = 5$ dan $y = 7$ maka ada dua pilihan yang dapat dipilih, yaitu pilihan kedua – dengan guard ($y > x$) – atau pilihan keempat – dengan guard ($x \neq y$). Pilihan akan dipilih secara random.

Jika semua pilihan tidak ada yang *executable*, maka proses akan terhambat (blocked) sampai ada salah satu yang bisa dieksekusi.

Pengulangan (*repetition*) dilakukan dengan keyword *do od*.

```

byte count;
proctype counter()
{
do
:: count = count +1
:: count = count -1
:: (count == 0) -> break
od
}

```

Di dalam PROMELA diperkenankan untuk melakukan *unconditional jump* dengan kata kunci *goto*. Flow akan diteruskan kepada bagian (label) yang dituju oleh *goto*. Dalam contoh di bawah ini apa bila nilai x sama dengan y maka flow akan diteruskan ke label “done” yang kemudian dilanjutkan dengan perintah *skip* yang tidak melakukan apa-apa.

```

proctype Euclid(in x, y)
{
do
:: (x > y) -> x = x - y
:: (x < y) -> y = y - x
:: (x == y) -> goto done
od
done:
    skip
}

```

2.3.2 Tool SPIN

SPIN merupakan sebuah tool yang dapat digunakan untuk melakukan *model checking* terhadap sistem yang ditulis dalam bahasa PROMELA. SPIN

ditulis dalam bahasa C dan tersedia *source code*-nya untuk sistem UNIX dan Windows. Ada juga *front end* grafis berbasis Tcl/Tk.

Untuk mengenal SPIN dan bahasa PROMELA, mari kita lakukan sebuah kegiatan laboratorium. Diasumsikan bahwa eksperimen ini dilakukan pada sebuah komputer UNIX.

Efek Variabel Global

Pada eksperimen ini kita akan melihat efek dari variabel global. Ketikkan teks di bawah ini dengan menggunakan sebuah editor. Beri nama berkas “global.spin”.

```
byte state=1;
proctype A() { (state == 1) -> state = state + 1; printf("A\n") }
proctype B() { (state == 1) -> state = state - 1; printf("B\n") }
init { runA(); run B(); }
```

Pada contoh di atas kita menggunakan variabel global yang bernama “state” dan menjalankan dua proses A() dan B() yang saling berebut untuk melakukan perubahan terhadap nilai variabel tersebut.

Jalankan SPIN terhadap berkas “global.spin” yang baru kita buat tersebut.

```
unix% spin global.spin
```

Perhatikan hasil eksekusi, khususnya nilai dari variabel “state”. Hasil eksekusi ternyata tidak dapat diramalkan. Variabel “state” bisa berakhir dengan nilai 0, 1, atau 2. Jelaskan mengapa hal ini bisa terjadi? Kemudian, berikan usulan untuk mengatasi ketidakpastian tersebut.

2.4 Higher Order Logic

2.5 Petri nets

Informasi mengenai petri nets dapat diperoleh dari berbagai sumber seperti:

Ada tools untuk mensimulasi dan menganalisa Petri nets. Salah satu tools yang cukup baik adalah Design/CPN. Sementara itu perbandingan antara Petri net tools dapat dilihat di

<http://home.arcor-online.de/wolf.garbe/petrisoft.html>

Tutorial mengenai Petri nets dapat dilihat di

http://www.iai.inf.tu-dresden.de/ms/lvbeschr/vwahl_petri.html

2.6 Synchronized Transition

2.7 Lain-lain: CCS, Estelle, Lotos, VDM, Z

Selain formalism yang dijelaskan pada bagian sebelumnya, ada beberapa notasi lain yang juga memiliki banyak pengikut. Notasi tersebut antara lain:

- Calculus of Communicating System (CCS) yang dikembangkan oleh R. Milner [9]

Bab 3

Formal Synthesis

Spesifikasi diimplementasikan melalui proses sintesa (*synthesis*). Proses sintesa dapat dilakukan secara manual atau secara otomatis. Pembuatan implementasi secara manual hanya cocok untuk desain yang tingkat kompleksitasnya tidak terlalu tinggi. Untuk sistem yang kompleks, kesalahan sangat mudah terjadi (*error prone*).

Proses sintesa yang otomatis disebut “automatic synthesis”. Untuk menjamin kebenaran proses sintesa, maka setiap langkah dari proses sintesa tersebut harus dibuktikan kebenarannya. Proses sintesa ini dilakukan secara bertahap dimana setiap tahap dilakukan pembuktian (*proof*) sehingga hasil sintesa akan benar karena *correct by construction*.

Formal synthesis ini biasanya berangkat dari spesifikasi yang dibuat secara formal, kemudian dilakukan proses *refining* sehingga diperoleh hasil yang lebih detail - menuju ke level abstraksi yang lebih rendah (implementasi). Proses ini dilakukan berulang-ulang sampai akhirnya menjadi implementasi yang diinginkan. Dalam desain VLSI, proses ini dilakukan sampai menjadi level layout dari transistor. Setiap langkah pada proses *refining* “dijaga” oleh sebuah mekanisme, misalnya dengan menggunakan sebuah *theorem prover* sehingga tidak dapat melakukan proses *refining* yang salah.

Bab 4

Formal Verification

Non-exhaustive testing can be used to show the presence of bugs, but never to show their absence. (E. W. Dijkstra)

The real benefit of verification schemes is not in proving anything absolutely but in pinpointing defects along its way. (Ivar Peterson in "Fatal Defect" quoting Peter Neumann)

Tujuan dari *formal verification* adalah **membuktikan** bahwa sebuah implementasi betul-betul mengimplementasikan apa-apa yang dijabarkan dalam spesifikasinya. Bagaimana kita tahu bahwa rangkaian yang kita buat merupakan sebuah *adder*, misalnya.

Kata kunci dari *formal verification* adalah "membuktikan". Berbeda dengan simulasi yang hanya mengambil beberapa data untuk menguji rangkaian, *formal verification* menggunakan metoda matematik untuk melakukan pembuktiannya. Ini adalah salah satu kunci dari metoda formal.

Formal verification dapat juga digunakan untuk menjabarkan kebenaran dalam proses translasi. Ketika kita melakukan proses translasi - misalnya dalam proses optimasi untuk menghasilkan rangkaian yang lebih irit dalam jumlah gerbang (*gates*) yang digunakan - bagaimana kita menjamin bahwa rangkaian yang dihasilkan oleh proses translasi ini masih memiliki fungsi yang sama (ekivalen) dengan rangkaian sebelumnya.

Formal verification dapat dilakukan dengan menggunakan:

- equivalence checker
- model checker
- theorem prover

4.1 Equivalence Checking

Equivalence checking dengan menggunakan *equivalence checker* membandingkan dua buah rangkaian dan membuktikan bahwa keduanya memiliki fungsi yang ekuivalen. Proses yang dilakukan mencoba menjawab pertanyaan: “Apakah perubahan yang telah saya lakukan terhadap desain ini mengubah fungsinya?” Pertanyaan ini dapat terjadi ketika kita melakukan proses optimasi terhadap rangkaian. Misalnya, kita ingin agar rangkaian lebih hemat dalam menggunakan daya dengan mengorbankan *performance*, maka kita ubah rangkaiannya dengan mengurangi beberapa komponen.

Proses *equivalence checking* ini dapat dilakukan dengan berbagai cara, mulai dari manual sampai ke otomatisasi dengan menggunakan *Ordered Binary Decision Diagram* (OBDDs).

4.2 Model Checking

Model checking menangani masalah yang berbeda dengan *equivalence checking*. Dalam model checking, kita ingin menguji apakah desain ini memiliki sifat (*properties*) seperti yang kita harapkan? Singkatnya kita ingin membuat sebuah model abstrak dari desain kita, kemudian kita mencoba membuktikan *properties* dari desain tersebut misalnya apakah akan terjadi *deadlock* atau *livelock*.

Sudah ada beberapa tools untuk melakukan model checking, akan tetapi kebanyakan masih berupa tools yang dikembangkan untuk model berukuran kecil sebagai riset di perguruan tinggi.

4.3 Theorem Proving

Metoda yang menggunakan pendekatan *theorem proving* dapat dikatakan duduk di atas kedua metoda yang telah dibahas sebelumnya. *Theorem proving* dengan menggunakan *theorem prover* memiliki *features* yang lebih komplis. Sayangnya teknologi *theorem proving* ini masih baru dan tools yang ada masih sulit untuk digunakan.

Theorem prover merupakan tools yang dapat membantu desainer untuk membuktikan *property* dari sebuah sistem. Dia menjaga agar pengguna tools tersebut tidak melakukan langkah-langkah yang salah, seperti misalnya mengganti sebuah komponen dengan komponen yang berbeda fungsinya. *Theorem prover* masih membutuhkan bantuan pengguna untuk mengarahkan atau mengambil langkah-langkahnya.

Tabel 4.1: Tabel kebenaran F_1 dan F_2

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

4.4 Contoh masalah

Mari kita ambil contoh dua buah fungsi F_1 dan F_2 sebagai berikut¹.

$$F_1 = \bar{A} B C + A \bar{B} \bar{C} + A \bar{B} C + A B \bar{C} + A B C \quad (4.1)$$

$$F_2 = A + B C \quad (4.2)$$

Apakah kedua fungsi tersebut ekuivalen? Bagaimana cara kita untuk membuktikan bahwa keduanya ekuivalen (atau tidak)? Ternyata ada beberapa cara. Mari kita uraikan satu persatu.

4.4.1 Truth table

Cara pertama adalah dengan menggunakan tabel kebenaran (*truth table*). Untuk kedua contoh di atas kita peroleh *truth table* sebagaimana ditampilkan pada tabel 4.1. Pada tabel tersebut dapat dilihat bahwa F_1 memiliki nilai yang sama dengan F_2 untuk setiap kombinasi A , B , dan C . Sehingga dapat disimpulkan bahwa F_1 dan F_2 adalah ekuivalen.

Penggunaan *truth table* ternyata memiliki kelemahan, yaitu jumlah tabel meledak secara eksponensial. Dapat kita lihat bahwa jumlah baris dari tabel bergantung kepada jumlah variabel dengan rumus 2^n , dimana n merupakan jumlah dari variabel. Sebagaimana dicontohkan di atas, ada 3 buah variabel (A , B , dan C) sehingga jumlah baris dari tabel adalah $2^3 = 8$. Dapat dibayangkan jika kita memiliki persamaan dengan 64 variabel. Sungguh sangat besar tabelnya sehingga metoda ini kurang praktis.

¹Persamaan F_1 dikenal dengan bentuk *sum of product* (SOP) dimana semua variabel muncul pada setiap *term*. Setiap *term* berisi hasil proses AND (*product*) dari semua variabel yang kemudian di-OR-kan (*sum*). Jadilah *sum of product*.

4.4.2 Karnaugh-map

Cara kedua adalah dengan menggunakan *Karnaugh-map* (K-map). K-map dapat dianggap sebagai visualisasi dari *truth table*, dimana setiap variable dipetakan dalam format dua dimensi. K-map kemudian dapat digunakan untuk menyederhanakan persamaan (dalam hal ini menyederhanakan F_2) sehingga dapat diperoleh persamaan yang paling minimal.

Tugas: buat K-map dari F_1 dan F_2 . Tunjukkan bahwa keduanya adalah ekivalen dengan melakukan reduksi di K-map.

Seperti halnya dalam penggunaan *truth table*, penggunaan K-map juga terbatas untuk jumlah variable yang sedikit. Maksimal yang dapat dilakukan dengan menggunakan K-map adalah 8 variabel.

Masalah lain yang juga dihadapi adalah hasil minimisasi dengan K-map dapat berupa persamaan yang terlihat berbeda akan tetapi tetap ekivalen. Dengan kata lain, hasilnya tidak *canonical*. Sebagai contoh, kedua fungsi di bawah ini (g dan h) dapat dihasilkan dari penyederhanaan melalui K-map. Keduanya ekivalen² akan tetapi tidak terlihat dengan mudah.

$$\begin{aligned} g &= A B + C D \\ h &= \overline{AB} \cdot \overline{CD} \end{aligned}$$

4.4.3 Aljabar Boolean

Cara ketiga adalah melakukan manipulasi persamaan F_1 dan F_2 dengan menggunakan aljabar Boolean. Berbeda dengan metoda sebelumnya, metoda aljabar ini bisa digunakan untuk persamaan dengan banyak variabel. Hanya memang proses manipulasi persamaan tersebut bisa melelahkan dan menjadi sumber kesalahan. Untuk mengurangi masalah tersebut bisa digunakan *theorem prover*.

Tugas: sederhanakan persamaan F_1 sehingga akhirnya adalah sama dengan F_2 . Tunjukkan setiap langkah yang anda lakukan.

Masalah lain yang dihadapi dengan penggunaan manipulasi aljabar juga sama dengan penggunaan K-map, yaitu bisa dihasilkan persamaan yang berbeda meskipun keduanya ekivalen. Hasilnya tidak *canonical*.

²Ekivalensi dapat dilakukan dengan menggunakan dalil De Morgan.

4.4.4 Propositional Tableaux

Cara keempat adalah dengan menggunakan yang disebut *propositional tableaux*. Metodologi ini dijabarkan dalam buku "A course in mathematical logic" karangan John Bell [1] pada halaman 25-34. Metodologi ini juga menggunakan cara visual dan cukup melelahkan untuk persamaan dengan banyak variabel.

Cara berikutnya adalah dengan menggunakan *Binary Decision Diagram* atau BDD yang akan kita bahas pada bagian berikut.

4.5 Binary Decision Diagram

Pada bagian sebelumnya telah diuraikan beberapa cara untuk melakukan manipulasi fungsi Boolean (*Boolean functions*). Akan tetapi seringkali sulit untuk mengotomatisasi proses manipulasi tersebut dengan menggunakan komputer.

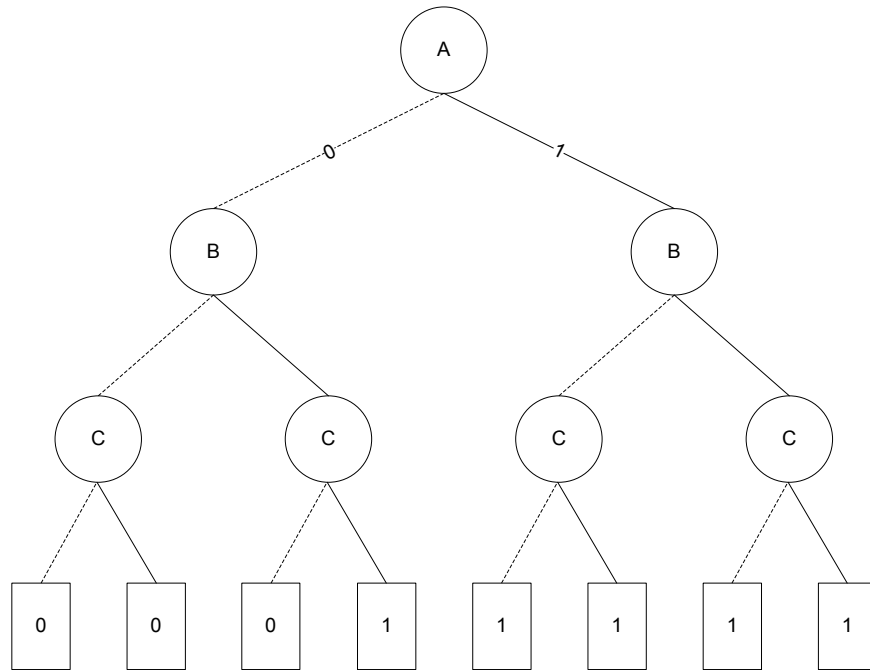
Binary Decision Diagram (BDD) merupakan sebuah representasi simbolik dari fungsi Boolean yang dikembangkan dan dipopulerkan oleh Randal Bryant dalam makalah klasiknya [2] dan *technical report* CMU [3]. Selain mengembangkan representasi simbolik tersebut, Bryant juga memberikan algoritma yang efisien untuk melakukan manipulasi terhadap BDD.

Uraian mengenai BDD kita mulai dengan sejarah penggunaan graf (*graph*) untuk merepresentasikan sistem biner. Ide penggunaan graf sudah dimulai oleh Lee dan Akers. Kemudian ide ini berkembang. Dalam dunia pemrograman, struktur data graf banyak juga digunakan untuk memecahkan berbagai masalah.

4.5.1 Binary Decision Tree

Kita mulai lihat penggunaan graf dalam merepresentasikan fungsi Boolean dengan *Binary Decision Tree*. Binary Decision Tree merupakan sebuah graf yang memiliki akar (*rooted*) dan memiliki arah (*directed*). Ada dua jenis vertex di Binary Decision Tree, yaitu *non-terminal* dan *terminal*. Terminal vertex adalah vertex yang tidak mempunyai cabang di bawahnya, atau tidak punya "anak" lagi. Sementara non-terminal vertex adalah vertex yang memiliki cabang (anak) di bawahnya.

Gambar 4.1 menunjukkan sebuah Binary Decision Tree yang merepresentasikan persamaan 4.1 dan 4.2 pada halaman 31. Pada gambar ini non-terminal vertex ditunjukkan dalam bentuk lingkaran, sementara terminal vertex dalam bentuk kotak.



Gambar 4.1: Sebuah Binary Decision Tree

Terminal vertex v diberi label $value(v)$, dimana nilainya adalah “0” atau “1”.

Non-terminal vertex v diberi label $var(v)$, dimana dalam hal ini dengan menggunakan variabel dari fungsi yang akan direpresentasikan oleh Binary Decision Tree ini. Dapat terlihat ada tiga variabel yang digunakan, yaitu A , B , dan C . Urutan dari variabel ini sembarang, akan tetapi nanti menentukan di OBDD.

Setiap non-terminal vertex memiliki dua buah cabang di bawahnya. Cabang ke kiri, $low(v)$, dikaitkan dengan kondisi bilamana variabel dari vertex tersebut (yaitu v) memiliki nilai “0”. Sementara itu cabang ke kanan, $high(v)$, dikaitkan dengan pilihan bilamana variabel vertex v bernilai “1”. Untuk memperjelas visualisasi, seringkali pilihan untuk nilai “0” (ke kiri, $low(v)$) digambarkan dengan garis putus-putus. Sementara pilihan lainnya digambarkan dengan garis penuh (solid).

Pada bagian bawah, terminal vertex mempunyai nilai ($value(v)$) sesuai dengan fungsi yang diimplementasikannya. Dalam contoh ini, nilai-nilai tersebut sesuai dengan persamaan 4.1 dan 4.2 pada halaman 31. Kita bisa uji kebenarannya dengan menelusuri (*traversing*) dari akar Binary Decision

Tree ini, yaitu dari vertex A , sampai ke terminal vertex. Jika variable v diberi nilai 0, maka vertex selanjutnya adalah yang berada di jalur $low(v)$. Jika variabel v diberi nilai 1, maka vertex selanjutnya adalah yang berada di jalur $high(v)$. Sebagai contoh, jika kita tetapkan $A = 0$, $B = 1$, $C = 1$, maka kita akan berakhir di terminal vertex yang bernilai “1” yang terletak nomor empat (4) dari sebelah kiri.

Binary Decision Tree ini memiliki kelemahan. Salah satu kelemahan yang paling utama adalah banyaknya sub-tree yang redundan sehingga membuat ukuran tree ini menjadi besar – meledak secara eksponensial terhadap jumlah variabel. Kita lihat untuk sub-tree yang bermula (berakar) di vertex C . Dalam contoh yang sederhana ini saja hanya ada tiga buah sub-tree C yang berbeda, yaitu yang anaknya $(0, 0)$, $(0, 1)$, dan $(1, 1)$.

4.5.2 Dari Binary Decision Tree ke BDD

Mari kita sederhanakan Binary Decision Tree kita. Ada tiga langkah yang akan kita lakukan:

1. menghilangkan terminal vertex yang redundan,
2. menghilangkan non-terminal vertex yang redundan,
3. menghilangkan test yang redundan.

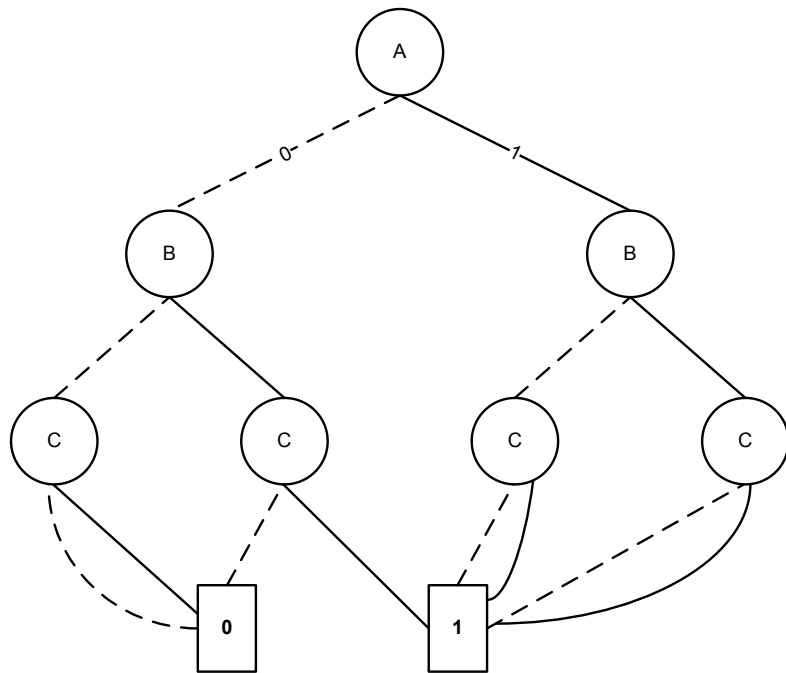
Menghilangkan terminal vertex yang redundan

Ada dua jenis terminal vertex dalam graf Binary Decision Tree kita, yaitu vertex yang memiliki nilai “0” dan “1”. Namun ada delapan buah vertex tersebut.

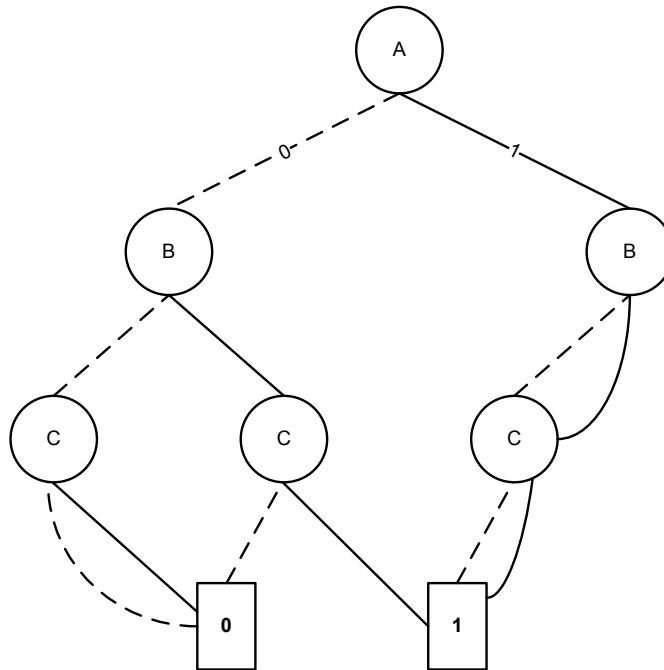
Kita sederhanakan graf kita dengan hanya menggunakan dua vertex saja, yaitu satu untuk vertex yang memiliki nilai 0 dan satu lagi yang memiliki nilai 1. Cabang yang sebelumnya mengarah ke vertex yang dihapuskan diarahkan ke vertex yang nilainya sama dengan vertex sebelumnya. Jika kita sederhanakan, grafiknya menjadi seperti yang ditunjukkan pada gambar 4.2. Langkah pertama selesai.

Menghilangkan non-terminal vertex yang redundan

Graf yang sudah kita sederhanakan tersebut masih bisa kita sederhanakan lagi dengan menghilangkan non-terminal vertex yang redundan (sama). Dua vertex dikatakan sama jika variabelnya sama, cabang ke kirinya sama, dan cabang ke kanannya juga sama. Secara formal, non-terminal vertices u dan



Gambar 4.2: Terminal vertex tidak redundan lagi



Gambar 4.3: Non-terminal vertex tidak redundan lagi

v dikatakan sama jika $var(u) = var(v)$, $low(u) = low(v)$, dan $high(u) = high(v)$. Dengan kata lain mereka merupakan sub-tree yang sama (isomorphism).

Pada gambar 4.2 terlihat ada dua buah non-terminal vertex yang redundan, yaitu dua vertex C di sebelah kanan. Kedua vertex tersebut mempunyai anak yang keduanya sama-sama menuju ke terminal vertex “1”. Kita bisa hapuskan salah satunya dan arahkan cabang yang menuju ke vertex yang dihapuskan ke vertex yang bertahan sehingga menghasilkan graf seperti yang ditampilkan di gambar 4.3. Cabang dari B yang tadinya menuju ke vertex yang dihilangkan dapat ditunjukkan ke vertex C yang tetap ada.

Sayangnya pada langkah ini kita hanya bisa mereduksi satu non-terminal vertex saja. Pada persamaan yang lain, yang lebih kompleks, langkah ini mungkin dapat menghapuskan banyak non-terminal vertex yang sangat menyederhanakan graf.

Menghilangkan test yang redundan

Langkah yang terakhir kita lakukan adalah menghapuskan test yang redundan. Test yang redundan ditandai dengan cabang kiri dan cabang kanan yang sama-sama menuju ke vertex yang sama. Dengan kata lain vertex v memiliki $low(v) = high(v)$.

Pada graf kita ada beberapa situasi seperti ini. Pada bagian kiri kita lihat ada vertex C yang kedua cabang di bawahnya sama-sama menuju ke terminal vertex “0”. Pada bagian kanan, kita lihat vertex B dan C yang juga layak disederhanakan.

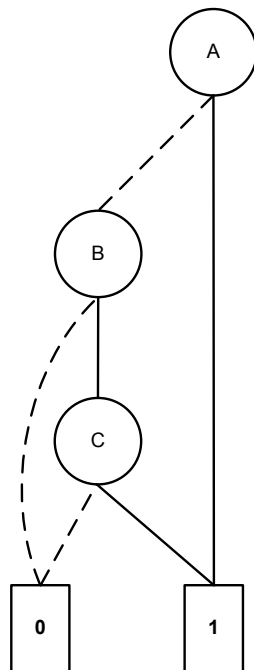
Proses penyederhanaan dilakukan dengan menghapuskan vertex tersebut dan mengarahkan cabang yang menuju ke vertex tersebut (*incoming arc*) ke vertex yang dituju oleh vertex yang akan dihapus tersebut. Jadi dalam contoh kita, pada bagian kiri, cabang sebelah kiri dari vertex B tidak lagi menuju ke vertex C di bagian kiri bawahnya, akan tetapi langsung menuju ke terminal vertex “0”. Demikian pula pada bagian kanan, cabang kanan dari vertex A tidak lagi menuju ke vertex B melainkan langsung menuju ke terminal vertex “1”. Hasil penyederhanaan ini dapat dilihat pada gambar 4.4. Ini yang disebut dengan Binary Decision Diagram (BDD) dari persamaan $F = A + B C$.

Proses penyederhanaan ini dapat dilakukan berulang-ulang sampai diagram tersebut tidak dapat direduksi lebih lanjut lagi. Randal Bryant dalam makalahnya membuat algoritma yang dia sebut *Reduce*, yang kompleksitasnya linear.

Tugas: buat BDD dari persamaan $G = A + B$, $H = A \cdot B$, dan $P = G + H$ (dimana G dan H merupakan hasil sebelumnya). Untuk fungsi P ini anda dapat menggabungkan fungsinya secara manual dahulu, baru kemudian dibuatkan BDDnya. Cara yang langsung untuk membuat BDD akan dijelaskan kemudian.

Jumlah vertex hasil penyederhanaan ini turun dari 15 vertex menjadi 5 buah. Selain itu diagram yang dihasilkannya ini bersifat *canonical*. Persamaan Boolean yang sama – dalam bentuk apapun awalnya – akan menghasilkan diagram yang sama. Jadi untuk menguji ekivalensi dari dua buah fungsi dapat dilakukan dengan menguji *isomorphism* dari dua buah BDD yang merepresentasikannya.

Perlu diperhatikan bahwa cara yang kita lakukan ini tidak efisien karena kita harus mulai dari graf yang lengkap, yang ukurannya eksponensial. Masalah lain yang kita hadapi adalah bagaimana kalau kita ingin menggabungkan dua BDD dalam sebuah operasi tertentu, seperti dicontohkan



Gambar 4.4: Hasil akhir penyederhanaan

dalam tugas. Untungnya Bryant [2, 3] sudah mengusulkan beberapa algoritma yang efisien untuk memanipulasi BDD.

4.5.3 OBDD

Secara formal BDD didefinisikan sebagai berikut [4].

A binary decision diagram is a root, directed acyclic graph with two types of vertices, terminal vertices and non-terminal vertices.

BDD memiliki sifat yang canonical jika urutan dari variable sama. Pada contoh sebelumnya kita menggunakan urutan variabel – dari atas ke bawah – A , B , dan kemudian C . Secara formal urutan ini disebut *variable ordering* dan dituliskan dengan $A < B < C$. Jika urutan ini kita ubah, maka hasil BDD yang diperoleh juga berubah.

Tugas: buat BDD dari persamaan $F = A + BC$ dengan *ordering* $C < B < A$. Bandingkan hasilnya dengan *ordering* sebelumnya $A < B < C$. Beri komentar mengapa ukuran graf BDD-nya berubah dan urutan mana yang lebih baik.

Untuk lebih eksplisit mengatakan bahwa kita hanya akan menggunakan BDD dengan *ordering* yang sama, maka kita definisikan *Ordered Binary Decision Diagram* (OBDD)³.

Seperti telah diutarakan sebelumnya, urutan dari variable menentukan ukuran dari BDD. Pertanyaannya adalah adakah algoritma untuk menentukan urutan yang efisien? Ternyata secara umum pencarian urutan yang optimal dikenal sebagai masalah yang memiliki kompleksitas *NP-complete*.

Biasanya urutan variabel yang baik bisa diperkirakan dari fungsi yang akan digunakan. Variabel yang berdekatan dalam fungsi tersebut sebaiknya diurutkan secara berdekatan juga dalam *ordering*-nya. Namun ini tidak selamanya dapat tercapai karena kadang-kadang dalam satu sistem ada banyak fungsi yang variabelnya bervariasi.

$$F = AC + BD \tag{4.3}$$

$$G = AD + BC \tag{4.4}$$

³Jika tidak membingungkan, seringkali saya menuliskan BDD meskipun yang dimaksud adalah OBDD.

Pada fungsi F kita berharap bahwa variabel A dan C berdekatan dalam urutannya. Sementara itu jika kita melihat fungsi G , maka kita berharap bahwa variabel A dekat dengan variabel D . Untuk kasus seperti ini tentunya kita hanya bisa memilih salah satu, C atau D sesudah A . Apapun pilihan kita, salah satu BDDnya tidak begitu efisien.

4.5.4 Operasi terhadap OBDD

Sebelum kita melakukan operasi terhadap BDD perlu kita definisikan beberapa hal. Kita mulai dengan fungsi yang membatasi (*restrict*) argumen x_i dari sebuah fungsi Boolean f dengan sebuah nilai konstan b .

Fungsi ini dituliskan sebagai $f|_{x_i \leftarrow b}$ dan memenuhi persamaan berikut

$$f|_{x_i \leftarrow b}(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, b, x_{i+1}, \dots, x_n)$$

Dengan kata lain, kita gantikan variabel x_i tersebut dengan konstan b .

Jika f direpresentasikan sebagai OBDD, maka OBDD dari *restriction* $f|_{x_i \leftarrow b}$ dilakukan dengan melakukan *depth-first traversal* dari BDD yang bersangkutan. Untuk semua vertex v yang memiliki pointer ke vertex w dimana $\text{var}(w) = x_i$, kita ganti pointer tersebut ke $\text{low}(w)$ jika $b = 0$ dan ke $\text{high}(w)$ jika $b = 1$. Jika hasilnya belum canonical, kita terapkan fungsi *Reduce* seperti sudah dijabarkan sebelumnya.

Semua logical operator yang menggunakan dua argumen (AND, OR, dan sejenisnya) dapat diimplementasikan secara efisien oleh OBDD. Dikatakan bahwa kompleksitas dari operasi ini linear terhadap ukuran dari argumen OBDD.

Kunci keberhasilan dari implementasi tersebut adalah adanya *Shannon expansion*:

$$f = \bar{x} \cdot f|_{x \leftarrow 0} + x \cdot f|_{x \leftarrow 1}$$

Shannon expansion memecah sebuah fungsi menjadi dua fungsi yang lebih sederhana. Sisi yang memiliki variabel \bar{x} terkait dengan cabang BDD yang berada pada sisi $\text{low}(x)$. Demikian pula sisi yang memiliki variabel x terkait dengan cabang yang berada pada sisi $\text{high}(v)$.

Apply

Operasi *apply* merupakan operasi yang menggabungkan dua buah BDD dengan sebuah operator biner (AND, OR, XOR, dan seterusnya sampai meliputi 16 operasi).

Ambil $*$ sebagai operator yang merepresentasikan operator biner, dan ambil f dan f' sebagai dua buah fungsi Boolean. Untuk menyederhanakan penjelasan dari algoritma, mari kita gunakan notasi berikut.

- v dan v' adalah akar dari OBDD f dan f' .
- $x = \text{var}(v)$ dan $x' = \text{var}(v')$.

Mari kita perhatikan beberapa kemungkinan yang bergantung kepada hubungan antara v dan v' .

- Jika v dan v' keduanya adalah terminal vertices, maka

$$f * f' = \text{value}(v) * \text{value}(v').$$

- Jika $x = x'$, maka kita gunakan *Shannon expansion*

$$f * f' = \bar{x} \cdot (f|_{x \leftarrow 0} * f'|_{x \leftarrow 0}) + x \cdot (f|_{x \leftarrow 1} * f'|_{x \leftarrow 1})$$

untuk memecah problem tersebut menjadi dua buah sub-problem. Kemudian sub-problem tersebut dipecahkan kembali secara rekursif.

Hasil dari proses ini adalah OBDD dengan akar v dimana $\text{var}(v) = x$

$\text{low}(v)$ merupakan OBDD dari hasil $(f|_{x \leftarrow 0} * f'|_{x \leftarrow 0})$

$\text{high}(v)$ merupakan OBDD dari hasil $(f|_{x \leftarrow 1} * f'|_{x \leftarrow 1})$

- Jika $x < x'$, maka $f'|_{x \leftarrow 0} = f'|_{x \leftarrow 0} = f'$ karena f' tidak bergantung kepada x . Jika demikian, maka Shannon Expansion menjadi

$$f * f' = \bar{x} \cdot (f|_{x \leftarrow 0} * f') + x \cdot (f|_{x \leftarrow 1} * f')$$

dan OBDD ini kemudian dihitung secara rekursif seperti kasus sebelumnya.

- Jika $x' < x$, maka komputasinya sama seperti kasus di atas.

Langkah-langkah di atas dapat diimplementasikan dalam sebuah program. Bahkan, dengan teknik pemrograman yang khas, kompleksitas dari algoritmanya bisa menjadi polinomial. Contoh trik yang dapat digunakan antara lain:

- menggunakan *hash table* untuk menyimpan hasil sub-problem yang sudah dihitung sebelumnya,
- sebelum melakukan komputasi secara rekursif, tabel tersebut dicek dahulu untuk melihat apakah sub-problem tersebut sudah berhasil dipecahkan sebelumnya,
- jika sudah, maka hasil dari tabel tersebut yang digunakan.

OBDD telah berhasil digunakan untuk sistem dengan jumlah vertex ratusan ribu.

Tugas

Buat OBDD dari $F = A$ dan $G = B$ dengan ordering $A < B$. Kemudian buat $H = F \cdot G$ dengan menggunakan teknik di atas, dimana operasi $*$ dalam operasi AND. Setelah itu buat OBDD dari $J = H + B$.

4.5.5 Tools

Saat ini sudah ada beberapa tools untuk BDD. Beberapa diantaranya didaftarkan di situs web www.bdd-portal.org. Materi dalam tulisan ini menggunakan tools `kbdd`. Mudah-mudahan di masa yang akan datang akan lebih banyak pembahasan lagi mengenai tools BDD.

4.6 Verification Tools

Adanya tools mendekatkan metoda formal kepada realitas sehingga betul-betul dapat dimanfaatkan dalam desain sesungguhnya, bukan desain dalam ukuran mainan saja. Dalam beberapa bagian telah disinggung beberapa tools. Berikut ini daftar beberapa tools yang telah tersedia.

- SPIN: PROMELA tools
- Formality
- HOL Theorem Prover
- BDD tools
- Larch Prover

Bab 5

Studi Kasus

Bab ini menjabarkan beberapa contoh penggunaan metoda formal. Beberapa contoh merupakan karya mahasiswa yang mengikuti kuliah metoda formal ini.

Lampiran A

Laboratorium BDD

Bagian ini berisi panduan untuk melakukan eksperimen (laboratorium) Binary Decision Diagram (BDD), atau lebih tepatnya *Reduced Ordered Binary Decision Diagram* (ROBDD).

Pada eksperimen ini kita akan menggunakan paket program KBDD yang dikembangkan oleh Karl Berry dari Carnegie Mellon University ¹. Program `kbdd` ini berjalan di sistem UNIX atau variannya. Untuk itu penguasaan penggunaan UNIX – meskipun minimal, seperti dapat melakukan `login`, mengedit berkas, menjalankan program, dan `logout` – merupakan syarat untuk melakukan eksperimen ini.

A.1 Masuk ke sistem UNIX

Langkah pertama yang anda lakukan adalah masuk ke sistem UNIX yang telah ditentukan dengan menggunakan *userid* dan *password* yang telah diberikan. Jangan mencoba-coba menggunakan id dan password orang lain.

```
Login: userid-anda
Password: *****

unix$
```

¹Kode dari program `kbdd` ini ada pada situs web dari kuliah saya. Jika anda membutuhkannya, silahkan kontak saya.

Tanda (*prompt*) “unix\$” atau “unix%” menunjukkan bahwa kita sudah masuk ke sistem UNIX.

Setelah masuk ke sistem UNIX, ketikkan “kbdd” untuk menjalankan program kbdd. Anda akan mendapatkan *prompt* “KBDD” seperti contoh di bawah ini. Jika anda tidak mendapatkan *prompt* tersebut, ada kemungkinan paket program kbdd tidak terpasang atau tidak berada dalam PATH anda. Tanyakan kepada administrator atau asistem lab, dimana proram kbdd berada.

```
uni% kbdd
KBDD:
```

A.2 Mencoba BDD

Program kbdd dapat dijalankan secara interaktif dengan mengetikkan perintah satu persatu dan mengamati hasilnya, atau dengan mekanisme *batch*. Dalam mekanisme *batch*, daftar perintah dimasukkan ke dalam sebuah berkas kemudian dijalankan kbdd dengan *standard input* (`stdin`) diarahkan ke nama berkas tersebut. Proses *batch* ini akan dijelaskan kemudian.

Pada percobaan pertama, ketikkan perintah ini satu persatu dan amati keluaran atau hasilnya. (Nanti ini harus anda laporkan sebagai bagian dari laporan lab.)

```
unix% kbdd
bool a b c
eval f (a+b)
bdd f
eval g (a&c)
bdd g
```

Baris pertama, dengan perintah `bool`, membuat urutan (*ordering*) dari variabel yang dalam hal ini membuat *ordering* $a < b < c$. Baris kedua mendefinisikan fungsi f yang merupakan **OR** dari variabel a dan b . Baris ketiga membuat representasi BDD dari fungsi f tersebut. Hal yang sama

dilakukan pada baris keempat dan kelima, hanya dengan fungsi yang berbeda.

Yang perlu anda perhatikan adalah tampilan dari perintah `bdd`. Karena keterbatasan layar komputer, tampilan dalam format teks yang kurang manusiawi.

Setelah itu, anda dapat mengeksplorasi perintah-perintah yang ada di paket `kbdd` lainnya, seperti antara lain:

```
sof f
satisfy f
sop g
satisfy g
```

Perintah `sop` memberi keluaran *Sum Of Product* dari fungsi yang bersangkutan. Sementara itu perintah `satisfy` memberikan daftar (set) dari input yang menghasilkan nilai “1” bagi sang fungsi.

A.3 Equivalency, isomorphism

Pada latihan kedua ini kita akan mencoba fungsi `kbdd` yang lebih spesifik untuk verifikasi. Perintah `verify` akan digunakan untuk menguji apakah dua fungsi yang diberikan adalah ekuivalen. Atau dengan kata lain terdapat isomorphism pada kedua graf OBDD dari kedua fungsi tersebut.

Pada contoh ini perintah-perintah `kbdd` dapat anda ketikkan dalam sebuah berkas yang kita beri nama “`latihan2.bdd`”. Kemudian kita jalankan `kbdd` dengan mengarahkan STDIN ke berkas tersebut. Ketikkan baris berikut.

```
echo Test of kbdd Boolean manipulator
bool a b c
eval f1 ((a+b)&c)
eval f2 (a&!c)
eval f f1+f2
eval g (a+(b&c))
echo These two functions should be equivalent:
verify f g
quit
```

Kemudian jalankan kbdd dengan cara berikut:

```
unix% kbdd < latihan2.bdd
```

Perhatikan makna dari setiap perintah dengan mengamati “KBDD Quick Reference” yang ada pada paket KBDD.

A.4 Efek dari ordering

Pada latihan ketiga ini kita akan melihat efek dari ordering pada sebuah fungsi. Fungsi yang akan kita uji memiliki persamaan sebagai berikut

$$f = (a_1 b_1) + (a_2 b_2) + (a_3 b_3) + (a_4 b_4)$$

Pada percobaan pertama kita akan menggunakan ordering sebagai berikut:
 $a_1 < a_2 < a_3 < a_4 < b_1 < b_2 < b_3 < b_4$

```
bool a1 a2 a3 a4 b1 b2 b3 b4
eval f ((a1&b1) + (a2&b2) + (a3&b3) + (a4&b4))
bdd f
quit
```

Pada percobaan kedua kita akan menggunakan ordering sebagai berikut:
 $a_1 < b_1 < a_2 < b_2 < a_3 < b_3 < a_4 < b_4$

```
bool a1 b1 a2 b2 a3 b3 a4 b4
eval f ((a1&b1) + (a2&b2) + (a3&b3) + (a4&b4))
bdd f
```

Perhatikan perbedaan dari bdd yang dihasilkan oleh kedua ordering di atas. Jelaskan dalam laporan mengapa hal ini terjadi.

A.5 Lain-lain

Paket KBDD ini dapat digunakan sebagai *library* untuk mengembangkan software sendiri yang dibuat dalam bahasa C. Misalnya, anda ingin membuat sebuah *logic simulator* dan menggunakan BDD untuk melakukan reduksi persamaan. Maka anda dapat gunakan library dari KBDD ini.

Daftar Pustaka

- [1] John Bell and Moshe Machover. *A Course in Mathematical Logic*. North-Holland, 1977.
- [2] Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.
- [3] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagram. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, July 1992.
- [4] Edmind M. Clarke. Model checking I: Binary decision diagram. In *Tutorial Notes FORTE'95*, Hilton, Montreal, Canada, October 1995. The 8th International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols.
- [5] C. A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21:666–677, August 1978.
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1995.
- [7] Greg Hoglund and Gary McGraw. *Exploiting Software: how to break code*. Addison Wesley, 2004.
- [8] Gerrard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [9] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. Lecture Notes in Computer Science vol. 92.
- [10] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Times Books, 1995.

- [11] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design*. Vintage Books, 1992.
- [12] Budi Rahardjo. Penggunaan formal methods dalam disain perangkat keras. Technical Report PPAUME-TR-1999-01, PPAUME ITB, January 1999.
- [13] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill International, fourth edition, 1999.
- [14] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., second edition, 1996.
- [15] Lauren Ruth Wiener. *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley, 1993.