論文99-36C-9-3

# Don't Care를 이용한 논리합성에서의 BDD 최소화 방법

## (BDD Minimization Using Don't Cares for Logic Synthesis)

洪 裕 杓 *, 朴 台 根 **

(Youpyo Hong and Taegeun Park)

요 약

BDD는 논리합성 응용분야에서 많이 이용되는 데이터 구조체이다. 불완전하게 표시된 함수를 합성할 때, BDD의 크기를 최소화할 수 있으면 BDD의 구조를 따라 만들어지는 회로의 크기나 동작 속도를 향상시키는데 큰 잇점이 있다. 본 논문에서는 논리합성를 위해 don't care를 이용하여 BDD를 최소화하기 위한 두 가지 알고리즘을 소개하고자 한다. 실험결과 제안된 방법은 기존 방법에 비하여 수행시간의 희생 없이 더욱 작은 크기의 BDD를 얻을 수 있었다.

**Abstract**

In many synthesis applications, the structure of the synthesized circuit is derived from its BDD functional representation. When synthesizing incompletely specified functions, it is useful to minimize the size of these BDDs using don't cares. In this paper, we present two BDD minimization heuristics that target these synthesis applications. Experimental results show that new techniques yield significantly smaller BDDs compared to existing techniques with manageable run-times.

## Ⅰ. 서 론

부울함수의 효율적인 표현은 논리합성, 테스트, 논리검증 등을 포함한 여러 CAD 분야에 필수적이다. BDD(Binary decision diagram)는 부울함수를 이용한 표현 및 조작에 효과적인 방법으로 알려져 있다[1]. BDD를 기초로 하는 방법에서 BDD의 크기는 실행시간은 물론 논리합성의 결과에도 매우 중요한 영향을 미친다.

현재 많은 논리 합성 소프트웨어가 BDD 함수표현 방법으로부터 직접 논리합성을 수행한다. 예를 들어, 멀티플렉서를 기초로 하는 무위험 다단(hazard-free multi-level) 논리회로가 BDD로부터 직접 합성되었고[2] T. Karoubalis[3] 등은 DCVS 회로가 BDD의 규칙성 때문에 BDD로부터 최적으로 합성될 수 있음을 보여주었다. 더욱이 Lavagno[4] 등은 BDD를 기본으로 하는 새논 회로합성 방법을 제시하며 BDD 크기가 소비전력의 감소를 유도했다는 결과를 보고하였다. 기술 매핑 분야에서 멀티플렉서형 FPGA 매핑은 BDD로 표현된 논리함수로부터 직접 실행될 수 있다[5,6]. Chang 등은 FPGA 매핑에서 BDD로 표현되는 그래프의 크기를 최소화하기 위하여 don't care를 이용한 BDD 최소화 방법을 제안하였다[6].

소프트웨어 합성 분야에서도 BDD를 이용한 방법이 응용되고 있다. Chiodo 등은 함수의 BDD 표현과 합성시 소프트웨어 프로그램의 긴밀한 연관성에 착안하여 소프트웨어 프로그램을 생성하기 위한 중간 표현 방법

* 正會員, 東國大學校 電子工學科
(Dept. of Electronics Engineering Dongguk University)
** 正會員, 카톨릭大學校 컴퓨터電子工學部
(Dept. of Computer and Electronics Engeering Catholic University)
※ 본 연구는 1999년 동국대학교 신임 교원 연구비 지원에 의해 이루어지었음.
接受日字 : 1999年5月3日, 수정완료일 : 1999年8月11日

으로 BDD를 이용하였다[7]. 이때 BDD의 크기는 소프트웨어의 크기와 매우 깊은 관계가 있다.

불완전하게 표현된 함수들을 BDD로 표현하기 위해서는 여러 개의 BDD가 필요한데 각각은 2진 값을 don't care(DC)로 할당하게 된다. 이 때, 최소화된 BDD를 얻기 위한 할당 방법을 찾는 문제는 NP-Complete[8]의 복잡도를 갖는다고 알려져 있고 이 경우 정확한 해를 찾는 것[9]은 비현실적이므로 많은 경험적 알고리즘들이 연구되었다. 이들 방법들은 최소화 연산 시 노드 공유(node-sharing)와 형제치환(sibling-substitution)[10]의 극대화를 추구하였다. BDD 노드들은 만일 DC의 재할당의 결과 그의 해당 함수가 동일하다면 공유된다. 형제치환은 노드공유의 특별한 경우인데 BDD 노드 u의 자녀(child)가 u의 다른 자녀에 의해 치환되어 u와 자녀가 제거되는 경우를 의미한다.

Chang 등은 BDD 트리를 위로부터 아래로 검색하여 각 단에서 다수의 노드를 공유하도록 하는 방법을 제시하였다[11]. 결과의 최소화 능력은 향상 되었으나 실행시간 또한 크게 증가하였다. restrict and constrain(또는 generalized-cofactor)[12,13]는 형제치환을 이용한 대표적 알고리즘이다. 이 방법의 문제는 결과 BDD가 본래의 BDD 보다 큰 경우도 있을 수 있다는 것이다. 이러한 경우를 피하기 위하여 thresholding이라는 방법을 제시하였지만 이 역시 문제를 완전히 해결하지 못하였다. 그리하여 안전(safe) BDD 최소화 알고리즘이 제시되었는데, 이는 BDD의 크기를 증가시키는 원인을제거하여 그 크기 증가를 원천적으로 방지함을 의미한다[14].

본 논문에서는 BDD 최소화를 위한 두 가지 안전 알고리즘을 제시하였다. 첫째 알고리즘은 형제치환 방법과 BDD의 크기증가를 억제하는 방법을 포함하고 있다. 둘째 알고리즘은 형제치환이 적용될 때[16]에서 제시한 BDD의 크기증가를 막는 좀 더 세심한 방법을 사용하였다. 모두 안전(safety)의 일반적인 개념을 이용하여 multi-rooted BDD의 경우를 해결하였다. 제시된 알고리즘은 ISCAS와 MCNC의 벤치마크 문제를 실험하여 기존 방법에 비하여 수행시간의 희생 없이 더욱 작은 크기의 BDD를 얻을 수 있음을 보여주었다.

2장에서는 관련된 이론을 정의하였고, 3장에서 두 가지의 새로운 방법을 논하였다. 4장에서는 실험결과를 분석하였고 결론은 5장에 제시하였다.

## Ⅱ. 배 경

노드집합 N과 에지집합 E로 나타내는 BDD[2]에서 N에 속하는 노드에는 종결노드(leaf node)와 비종결노드(non-leaf node)의 두 가지로 나눌 수 있다. 종결노드는 부울함수 0이나 1의 값을 갖으며 각 비종결노드 u는 두 개의 밖으로 나가는 에지 즉, then_edge와 else_edge를 갖는다. 각 에지는 u의 자녀노드와 연결되어 있고 u는 그 자녀노드의 부모노드라고 한다. 두 개의 자녀노드들을 형제노드(siblings)라고 한다. 비종결노드 u는 level(u)라는 정수 값을 갖고 u, v N일 때, 각 에지 (u, v) E는 level(u) < level(v)를 만족한다. 비종결노드 F는 순환적으로 정의되는 부울함수이다. 함수 F는 x=1일 때 then_edge를 통해 자녀노드로 연결되는 함수 Fx와 x=0일 때 else_edge를 통해 자녀노드로 연결되는 함수 $F_{x'}$로 정의된다.

단일 출력을 갖는 부울함수 ff는 $ff_{on}$(on-set), $ff_{off}$(off-set), 그리고 $ff_{dc}$(DC set)의 세 가지의 값을 갖는다. 이들 중 어느 두 가지만으로 부분적(incompletely)으로 표현된 함수를 정확히 나타낼 수 있다. 불완전하게 표현된 함수 ff는 $f_{on} \supseteq ff_{on}$, $f_{Off} \supseteq ff_{off}$, 그리고 $c = ff_{on} \cup f_{off}$를 만족하는 완전히 표현된 함수 [f, c]의 쌍으로 표현될 수 있다[10]. [f, c]는 일반적으로 [F, C]의 BDD 쌍의 형식으로 주어진다. 주어진 [F, C]에서 cover F'을 찾는 작업을 BDD 최소화 작업이라고 하고 이 때, F는 초기 BDD, F'은 최소화된 BDD라고 한다.

**Definition 1**[10] *불완전하게 표현된 함수 ff에 대하여 $f_{on} \supseteq ff_{on}$, $f_{off} \supseteq ff_{off}$을 만족할 때 f는 ff의 cover라 말한다.*

## Ⅲ. 최소화 알고리즘

### 1. 형제치환의 원리와 이를 이용한 BDD 최소화

형제치환(sibling-substitution)이란 비종결노드 u와 그의 자녀 v, w가 있을 때 v를 w로 치환하는 작업이다(결국 u는 w로 치환된다). 형제치환이 노드 u에 적용될 때 u는 그의 자녀노드에 의해 치환될 부모노드가 된다. 형제치환을 이용한 BDD 최소화 방법들의 차이점은 형제치환이 적용되는 경우를 결정하는데 있다. F에

속한 에지와 C에 속한 노드간에 매핑에 관해 살펴보면, F의 노드 u의 then_edge(else_edge) e가 C의 노드로 매핑될 때 map(e)라고 한다. 변수 x가 노드 u와 연관되고 x=1(x=0)에 대한 부분 입력조합 p와 그의 확장형태 p이 존재한다면 v∈map(e)이다. 단, 함수 F가 p에 대하여 u를 p에 대하여 $u_x(u_{\bar{x}})$를 연산하며 함수 C는 p에 대하여 v를 연산하는 관계에 있다. 그림 1의 (a), (b)에 있는 F와 C를 가정해 보자. 부분 입력조합 a=0의 값은 F와 C의 노드 b를 연산한다. 그리하여 F에 있는 a의 else_edge는 C에 있는 b로 매핑 된다. 유사하게 부분 입력조합 a=1의 값은 F의 노드 c와 C의 종결노드 1를 연산한다. F의 각 에지에 대한 노드 매핑은 그림 1(c)에 나타내었다. 그림에서 then_edge (else_edge)는 실선(점선)으로 나타내었고 에지 위의 1(0)은 종결노드 1(0)를 나타낸다.

Restrict는 F와 C를 순환적으로 수행한다. map(e)는 에지 e가 분석되는 여러 순환연산 동안에 다루어지는 care노드의 집합이다. e가 노드 u에서 나가는 에지라고 가정하자. restrict는 만일 e의 care노드가 DC라면 u에 형제치환을 적용한다. 이는 수학적으로 DC map(e)임을 말한다. 만약 map(e)가 non-DC를 포함한다면 초기 노드가 최종결과에 필요하므로 문제를 일으키게 된다. 결과적으로 u와 같은 공유노드는 최소화 작업 동안에 공유되지 않게 된다. 이를 node-splitting이라고 말하며 이런 이유로 restrict가 BDD 크기를 증가 시키는 요인으로 작용한다.

B-compaction(Basic Compaction)과 LI-compaction(Leaf-Identifying Compaction)은 앞에서 설명한 안전[1]한 형제치환을 이용하여 BDD를 최소화하는 방법들이다. B-compaction은 만일 에지 e가 DC-leaf로 매핑되면 즉, map(e) = {DC}이면 형제치환을 적용한다. 각 에지에 대한 매핑노드의 구분은 mark-edges라는 루틴이 담당하는데, 이는 동시에 F와 C를 검색하여 C에 있는 노드가 DC-leaf인지 아닌지 분석한다. 그림 1은 B-compaction의 예를 보여준다. DC-leaf가 아닌 것으로 매핑된 에지들은 그림 1(d)에 점으로 표시하였다. build-result라는 루틴에서는 F에 표시된 에지에 대해서만 BDD F를 재구성한다. 만일 노드 v에서 그의 자녀노드 u로 가는 에지가 마크되어 있지 않다면 v는 u의 형제에 의해서 안전하게 치환될 수 있다. 그렇지 않다면 v는 보존되고 그의 자녀들은 순환적으로 재구성된다. 그림 1(e)는 build-result의 결과를 나타내었다.

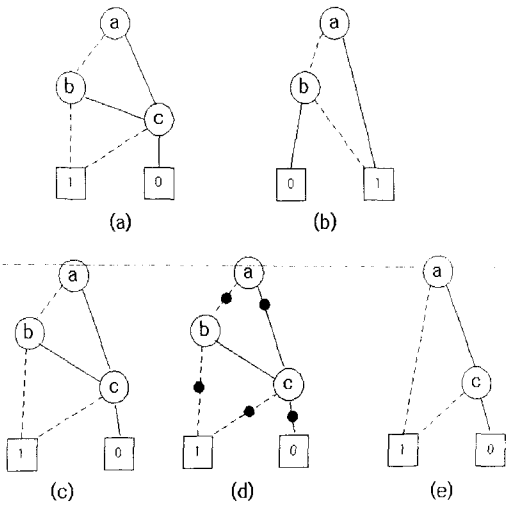



그림 1. (a) F : 초기 BDD (b) C : Care-BDD (c) 매핑노드에 의해 라벨이 붙여진 F의 에지 (d) Edge-marked F (e) B-compaction 결과

Fig. 1. (a) F : BDD to minimize (b) C : Care-BDD. (c) Edges in F labelled by mapping nodes (d) Edge-marked F (e) B-compaction result

두 번째 BDD 최소화 방법인 LI-compaction은 종결-BDD를 제외한 모든 BDD에 대하여 종결노드의 필수성을 분석함으로써 B-compaction의 방법을 개선한 것이다. 중요한 개념은 만일 자녀노드와 연결된 에지가 종결노드로 바로 갈 수 있다면 그 노드에 대해 형제치환을 적용한다. 이러한 방법은 전체 BDD 크기를 증가시키지 않고 최소화하는데 도움을 준다.

### 2. 일반적 치환을 이용한 BDD 최소화 (General-Substitutability-Based Compaction)

Restrict는 자녀노드 중의 하나가 DC에 해당할 때 형제치환을 적용한다. Shiple[10]등은 이 조건이 형제치환을 위한 충분하지만 필요조건은 아님을 지적하였다. 본 논문에서는 BDD 크기증가를 억제하고 선별적으로 형제치환을 진행하는 일반적인 방법을 제안하는 첫 단계로 아래와 같이 두가지 치환에 대한 정의를 내린다.

***Definition 2 일반적-치환성*** *만일 $f_2$가 $[f_1, c_1]$의 cover라면 $f_1$은 $f_2$로 치환될 수 있다.*

***Definition 3 DC-치환성*** *만일 $c_1$이 $f_1$의 care함수일 때 $c_1$=0라면, $f_1$은 $f_2$에 의해 DC치환 가능하다라고 한다.*

일반적 치환성은 보다 광범위한 치환을 허용하며, 이에 근거하여 치환의 방향을 계산하기 위한 가코드(pseudo-code)는 그림 2에 나타냈다. 그림 3은 치환기준의 중요성을 보여주는데 형제치환은 정의 2, 3에 의하여 각각 a 혹은 b에 적용된다. 결과적으로는 일반적 치환성을 이용하여 a에 적용된 것이 더 작고, 이는 더 상위의 노드(더 큰 sub-BDD)에 적용해서 더 많은 노드들을 제거할 수 있었기 때문이다.

```
/* substitutability returns f_x tof'_x if f_x is substitutable
   by f'_x, f'_x tof_x if f'_x is substitutable by f_x, NONE
   otherwise */
int substitutability(bdd f, bdd c) {
  if(c==bdd_one)then return(NONE);
  x=top_variable(f,c);
  if(c_x=bdd_zero) then return(f_x tof'_x);
  if(c'_x==bdd_zero) then return(f'_x tof_x);
  f_diff=bdd_xor(f_x, f'_x);
  if(bdd-intersect-empty(f_diff, c_x)==TRUE)return(f_x tof'_x);
  if(bdd-intersect-empty(f_diff, c'_x)==TRUE)return(f'_x tof_x);
  /* bdd-intersect-empty returns TRUE if the conjunction
     of argument BDDs is bdd_zero */
```

그림 2. 치환성 검사 가코드
Fig. 2. Substitutability check pseudocode.

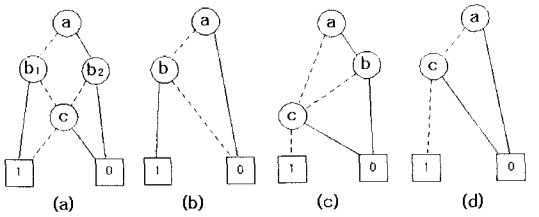



그림 3. (a) F (b) C (c) DC-치환성에 의한 $b_1$의 형제치환 (d) 일반적 치환에 의한 a의 형제치환
Fig. 3. (a) F (b) C (c) Sibling-substitution of $b_1$ using DC-substitutability (d) Sibling-substitution of a using DC-substitutability.

B-compaction에 일반화된 기준을 적용하기 위해 먼저 단순히 DC-치환성의 기준을 바꾸는 간단한 방법을 생각해 보자. 그러나 이는 다음과 같은 문제가 있다. B-compaction에서 에지는 만일 그 에지가 어떤 노드로 재설정될 수 있다면 즉, DC로 매핑 된다면 마크되지 않는다. 그러나 이는 edge-marking을 이용하는 치환에서는 사실이 아니다. 노드 u에서 v로 가고 care노드 $\{c_1, c_2\}$로 매핑되는 에지 e를 생각해 보자. v가 그의 형제에 의해 치환될 수 있을 때, edge-marking 작업은 순환하지 않으므로 e와 v 밑에 있는 모든 에지들은 결과적으로 마크되지 않는다. 그리하여 map(h)는 non-DC를 포함하고 v 밑에 있는 에지 h가 존재한다. $c_2$를 분석할 때, 에지 e가 마크된다면 문제가 발생한다. build-result 루틴에서 u의 형제치환은 e가 마크되었기 때문에 수행되지 않는다. 이는 더 이상 u의 cover가 아닌 BDD를 만든다. 예를 들어 그림 4에서 F에 속한 노드 c의 then_edge는 c가 그의 형제에 의해 치환 가능하기 때문에 그것이 care-node로 매핑된다 할지라도 마크되지 않는다. F에 속한 노드 c가 C의 노드 $c_2$를 만날 때, b의 else_edge는 마크된다. 결과적으로 build-result 과정은 그림 4(e)에서 보는 바와 같이 then_edge에 연결된 자녀를 치환할 때 틀린 결과를 얻는다. 그러므로 에지에 마크를 하지 않는다면 의도적인 형제치환을 의미한다. 노드 v가 마크되지 않은 에지와 연결되어 있을 때 v가 그의 형 구성되는데, 하나는 현재 마크되지 않은 에지는 계속 마크되지 않을 것이라고 가정하여 mark-edge를 수행하는 것이고 또 다른 하나는 처음 부분에서 수행된 형제치환이 옳은지 분석하고 아니라면 다시 형제치환을 수행하는 부분이다.

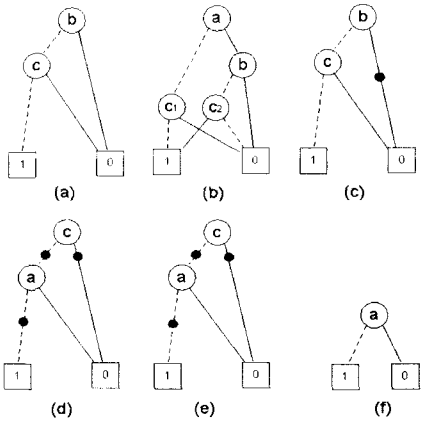



그림 4. B-compation에서의 일반치환성 사용 (a) F (b) C (c) $c_1$에 의한 F에지 마칭 (c) $c_2$에 의한 F에지 마칭 (d) 마킹 결과 (f) 최소화 결과
Fig. 4. General-substitutability in B-compation (a) F (b) C (c) edge-marking using $c_1$ (c) F edge-marking result $c_2$ (d) Final edge-marking result (f) Minimization result.

새로운 최소화 방법인 GS-compaction (Generalized Substitutability based Compaction)에 대한 가코드(pseudo code)는 그림 5에 나타내었다. 첫번째 에지 마킹 부분

인 mark-essential edges는 분석 중인 노드와 연결된 에지의 마크상태를 보고 형제치환의 수행조건을 테스트한다. 만일 조건을 만족시키면 그 노드의 형제들과 연결된 에지를 마크하고 마크된 에지와 연결된 노드에 대하여 계속 수행한다. 형제치환은 F와 C의 노드 그리고 치환의 방향 등을 포함하는 리스트에 기록된다. 그림 4(e)에 mark-essential-edges의 결과를 나타내었다. 두번째 에지 마킹 부분인 mark-supplemental-edges는 먼저 상위의 F 노드와 연결된 에지에 대해 수행하기 위하여 리스트를 F 노드의 레벨에 의해 재배열한다. 그리고 리스트 내에의 형제치환이 옳은지 분석한다. 만일 옳으면 치환될 노드의 care-set이 그의 형제노드의 care-set에 더해진다. 그렇지 않으면 F와 C노드에 대한 mark-essential-edges를 수행한다. mark-essential-edges가 끝나면 mark-supplemental-edges가 리스트가 빌 때까지 계속된다. build-result 부분은 B compaction 부분과 동일하다.

Mark-essential-edges의 복잡도는 치환부분이 O(|F||C|)이고 mark-essential-edges가 최대 |F||C|번 수행되므로 결국 O((|F||C|)2)이다. 재배열에 해당된 시간을 고려하면 O((|F||C|)2 + |F||C|log(|F||C|))이다. mark-essential-edges는 cache를 이용하면 같은 것을 반복 수행하지 않고 build-result와 clear-edge가 O(|F||C|)의 복잡도를 갖으므로 전체는 최종 O((|F||C|)2)이다.

```
bdd GS-compaction (bdd f, bdd c) {
   if (c == bdd_zero) return (bdd_zero)
   fcs_list = create_list();
   Mark-essential-edges(f, c, fcs_list);
   Mark-supplemental-edges(fcs_list);
   clear-edges();
   return(GS-build-result(f));
}
void Mark-essential-edges(bdd f, bdd c, list fcs_list) {
   if (c == bdd_zero || f == leaf) return;
   x = top_variable(f, c);
   if (f != fx)
      s = substititutability(f,c);
   if(f.then_edge ==0 )
      if(s == fxtof'x) insert(fcs_list,{f,c,s});
      else f.then_edge = 1;
   if(f.else_edge == 0)insert(fcs_list,{f,c,s});
      if(s == f'xtofx== 0)
      else f.else_edge = 1;
   if(f.then_edge == 1||f == fx)
         Mark-essential-edges(f'x,c'x,fcs_list);
   if(f.else_edge == 1||f == fx)
         Mark-essential-edges(fx,cx,fcs_list);
}
void Mark-supplemental-edges(list fcs_list) {
   while(fcs_list not empty) {
      sort fcs_list by level of f in each triple;
      fcs = first(fcs_list);
      remove(fcs, fcs_list);
      f=fcs.f; c=fcs.c; s=fcs.s;
      x=top_variable(f,c);
      if(s==fxtof'x);
         if(f.then_edge==0)
              Mark-essential-edges(f'x,cx,fcs_list)
         else Mark-essential-edges(fx,cx,fcs_list);
      if(s==f'xtofx)
         if(f.else_edge==0)
              Mark-essential-edges(fx,c'x,fcs_list)
         else Mark-essential-edges(f'x,c'x,fcs_list);
       }
}
bdd GS-build-result(bdd f) {
   if(f=leaf) return(f);
   x=top_variable(f);
   if(f.then_edge==1 && f.else_edge==1)
     return bdd-find(x,build-result(fx),build-result(f'x));
   else if(f.then_edge==1&& f.else_edge==0)
    return (build-result(f'x);
   else
     return (build-result(fx);
```

그림 5. GS-compaction 가코드
Fig. 5. GS-compaction pseudo-code.

### 3. 필수노드 검색을 이용한 방법 (Essential-Nodes Identifying Compaction)

이번 절은 LI-compaction을 개선한 방법이다. LI-compaction은 BDD의 cover를 위한 필수요소로서 종결노드만을 고려한다. 개선의 중요한 포인트는 필수노드의 더 큰 부분집합을 찾아내는 것이다. 어떠한 노드가 leaf1map(e)의 관계를 만족하는 에지 e에 연결되었을 때 그 노드는 필수노드라 한다. 이는 e에 의해 지정되는 노드 u로 표현되는 함수를 필요로 하는 입력조합이 적어도 하나 존재한다는 뜻이다. 그래서 u로부터 근원되는 부분 BDD는 BDD의 어떤 cover내에 있어야 한다. 필수노드를 찾아내기 위해 F와 C를 동시에 검색한다. C에 있는 종결노드1(leaf1)이 에지 e와 관련될 때, e에 의해 지정되는 노드는 필수적이라고 마크된다. 이 부분을 find-essential-nodes라고 하고 가코드는 그림 6에 있다.

```
void find-essential-nodes(bdd f,bdd c,hash_table H) {
   if(f==leaf)
      insert f in H;
```

```
    return;
  insert bdd_one in H;
  insert bdd_zero in H;
  find-essential-node-sub(f,c,H);
}
void find-essential-node-sub(f,c,H) {
  if(c==bdd_zero || f==leaf) return;
  if(f is in H) return;
  if(c==bdd_one)insert f in H;
  x=top_variable(f,c);
  find-essential-node-sub(fx, cx, H);
  find-essential-node-sub(f'x, c'x, H);
```

그림 6. Find-essential-node의 가코드
Fig. 6. Find-essential-node pseudo-code.

그림 7은 필수노드를 찾아내는 예이다. (a)와 (b)에서 보는 바와 같이 주어진 F와 C에서 각 에지에 대한 매핑노드들은 (c)에 나타내었다. (d)에서 $d_1$, $d_2$, 1, 0이 필수적이라고 분석되었다. 필수노드를 찾은 후에 edge-marking 부분이 각 F노드에 대한 DC치환성을 이용해 형제치환을 수행하고 치환될 자녀가 연결된 에지에 대한 결과를 저장한다. 만일 에지에 대한 결과가 독특하

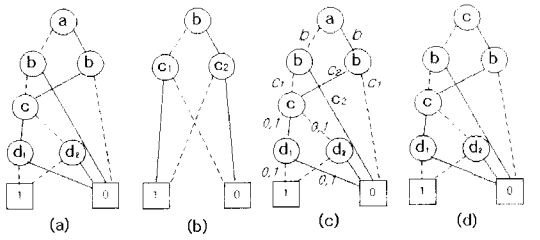



그림 7. 필수노드찾기 (a) F (b) C (c) 매핑노드에 의해 이름지어진 F 에지 (d) F에 마크된 필수노드
Fig. 7. Finding essential nodes (a) F (b) C (c) Edges in F labelled by mapping nodes (d) Essential nodes check-marked in F.

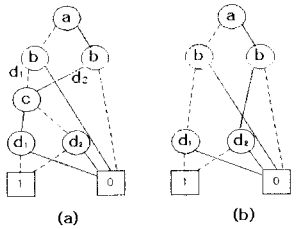



그림 8. EI-compction (a) 필수노드인 경우 restrict에 의한 에지들 (b) EI-compaction 결과
Fig. 8. EI-compction (a) Edges annotated *restrict* result if it is an essential node (b) EI-compaction result.

고 필수적이라면 그 에지를 필수노드로 재설정한다. 재설정은 BDD의 크기증가를 방지하는데 이는 필수노드는 초기상태를 포함한 어느 cover에도 존재하기 때문이다. 그림 8은 EI-compaction(Essential-Node Identifying Compaction)의 예이고 그에 대한 가코드는 그림 9에

```
bdd EI-compaction(bdd f, bdd c)
  if(c=bdd_zero) return(bdd_zero);
  H=create_hash();
  find_essential_nodes(f,c,H);
  (void)EI-mark-edges(f,c,H);
  result=EI-build-result(f);
  clear-edges(f);
  return(result);
}
/*   EI-mark-edges   returns   the   result   of
sibling-substitution if it is unique essential node. If
not, return INVALID symbol, DC is a default mark on edges
*/
bdd EI-mark-edges(bdd f, bdd c, hash_table H) {
  if(c==bdd_zero) return(DC);
  if(f==bdd_zero||f=bdd_one)return(f);
  x=top_variable(f,c);
  temp1=EI-mark-edges(fx, cx);
  temp2=EI-mark-edges(fx, cx);
  if(cx!=bdd_zero)
    if(temp1 is not in H) temp1=INVALID);
    else if(f.then_mark!=DC && f.then_mark!=temp1)
        temp1=INVALID;
  if(cx!=bdd_zero)
    if(temp2 is not in H) temp2=INVALID;
    else if(f.else_mark!=DC && f.else_mark!=temp2)
        temp2=INVALID;
  if(f!=fx)
    if(cx!=bdd_zero) f.then_mark=temp1;
    if(cx!=bdd_zero) f.else_mark=temp2;
  if(cx=bdd_zero) return temp2;
  else if(cx==bdd_zero) return temp1;
  else if(temp1==INVALID||temp2==INVLALID)return INVALID;
}
bdd EI-build-result(bdd f) {
  if(f=leaf) return(f);
  x=top_variable(f);

  if(f.then_mark==INVALID)f_left=EI-build-result(fx);
  else if(f.then_mark==DC) f_left=EI-build-result(fx);
  if(f.else_mark==INVALID)f_right=EI-build-result(fx);
  else if(f.else_mark==DC) f_right=EI-build-result(fx);

  if(f.then_mark==DC and f.else_mark!=DC) return f_right;
  else  if(f.then_mark!=DC  and  f.else_mark==DC)  return
f_left;
  else return(bdd_find(x, f_left, f_right)
}
```

그림 9. EI-compaction 가코드
Fig. 9. EI-compaction pseudo code.

나타내었다. EI-compaction의 복잡도는 LI-compaction과 같은 O(|F||C|)이다.

## Ⅳ. 실험 및 결과분석

우리는 제안된 두 가지 알고리즘을 B-compaction과 동시에 발표되어 그보다 우수한 성능을 보인 LI-compaction[14]과 비교하였다. 모든 실험은 SIS-1.2[15]에서 행해졌고 우리는 ISCAS-89와 MCNC-91의 벤치마크 회로에 무작위 DC를 이용해 실험하였다. 실험에서 10%와 90%의 DC fraction을 사용하였다.

표 1. 최소화 결과 (**BDD**: 초기 BDD 크기, **Sift**: sifting, TO: thresholded osm_bt, **LI**: leaf-identifying compaction, **GS**: general-substitutability based compaction, **EI**: essential node identifying compaction, **timeout**: 1시간 이상. 괄호 안의 숫자는 CPU 시간을 나타냄.)

Table 1. Minimization results (**BDD**: original BDD size, **Sift**: sifting, TO: thresholded osm_bt, **LI**: leaf-identifying compaction, **GS**: general-substitutability based compaction, **EI**: essential node identifying compaction, **timeout**: longer than 1 hour. Numbers in parenthesis show CPU sec.).

| 회로 | BDD | DC fraction : 10% | | | DC fraction : 90% | | |
|---|---|---|---|---|---|---|---|
| | | LI | GS | EI | LI | GS | EI |
| 9sym | 25 | 25 (0.02) | 24 (0.08) | 25 (0.06) | 25 (0.03) | 25 (0.08) | 23 (0.06) |
| z4ml | 47 | 15 (0.11) | 15 (0.08) | 15 (0.06) | 47 (0.08) | 44 (0.06) | 47 (0.04) |
| misex2 | 136 | 58 (0.37) | 53 (0.15) | 59 (0.13) | 131 (0.35) | 125 (0.11) | 131 (0.08) |
| s344 | 206 | 132 (0.2) | 117 (0.13) | 136 (0.08) | 205 (0.02) | 199 (0.11) | 201 (0.11) |
| s386 | 281 | 130 (0.25) | 92 (0.12) | 135 (0.08) | 280 (0.25) | 269 (0.12) | 269 (0.9) |
| duke2 | 973 | 211 (0.7) | 204 (0.12) | 214 (0.13) | 972 (0.36) | 956 (0.33) | 965 (0.2) |
| vg2 | 1,044 | 274 (0.27) | 275 (0.13) | 269 (0.11) | 1,044 (0.26) | 1,038 (0.43) | 1,044 (0.28) |
| misex3 | 1,301 | 567 (0.36) | 584 (0.19) | 565 (0.18) | 1,318 (0.47) | 1,206 (0.36) | 1,177 (0.29) |
| c432 | 1,733 | 1,395 (0.4) | 1,278 (1.45) | 1,313 (0..52) | 1,733 (2.2) | 1,732 (7.3) | 1,732 (3.5) |
| c1908 | 36,007 | 31,093 (6.5) | 23,150 (38.1) | 25,358 (14.8) | 36,007 (119) | 36,001 (387.2) | 36,001 (214.3) |

결과는 표1에 요약하였다 LI-compaction, GS-compaction, EI-compaction의 BDD 최소화 결과는 각각 LI, GS와 EI로 표시되었다. 다수 개의 출력을 갖는 BDD의 경우의 안전성에 대한 새로운 방법은 GS-compaction와 EI-compaction에만 적용되었다. 표는 GS-compaction이 가장 좋은 결과를 보여주었다. GS는 LI에 비하여 DC fraction이 90%일 때 약 10%정도 우수했고 DC fraction이 10%일 때 2%정도 우수했다. EI는 CPU시간에서 GS보다 적게 걸렸고 질적으로 경쟁력이 있었다. 그러나 그들간의 실행 시간차는 특히 90% DC에서 complexity와 완전 일치하지 않았는데, 이는 높은 DC fraction으로 인해 GS의 marking 대상이 적고, 상대적으로 EI는 필수 노드파악에 많은 시간이 소요 되었기 때문인 것으로 추정 된다.

GS는 치환성을 분석해야 하기 때문에 다른 방법들보다 수행시간이 더 소요되었다. 이는 GS에서 edge-marking부분이 불필요한 치환성 분석을 미리 제거해 주기 때문에 결과적으로 더 적은 치환성 분석이 수행되기 때문이다.

## Ⅴ. Conclusion

본 논문에서 새로운 BDD 최소화에 대한 두 가지 경험적 접근을 소개하였다. 새로운 방법은 여러 가지 예에서 종래의 방법에 비하여 우수한 결과를 보여 주었다. 본 논문의 최소화 방법과 형제치환에 근거하지 않은 다른 방법 과의 결합은 유망한 연구 과제로 고려될 수 있을 것이다.

## 참 고 문 헌

[1] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Computers, vol. C-35, pp. 677-691, 1986.

[2] B. Lin and S. Devadas, Synthesis of Hazard-Free Multi-level Logic under Multiple-Input Changes from Binary Decision Diagrams, Proc. International Conference on Computer-Aided Design, pp. 542-549, 1994.

[3] T. Karoubalis, G. P. Alexiou and N. Kanopoulos, Optimal Synthesis of Differential Cascode Switch Logic Circuits Using Ordered Binary Decision Diagrams, Proc. European Design

Automation Conference, pp. 282-287, 1995.

[4] L. Lavagno, P. McGeer, A. Saldanha and A. L. Sangiovanni-Vincentelli, Timed Shannon Circuits: A powerful Design Style and Synthesis Tool, Proc. Design Automation Conference, pp. 254-260, 1995.

[5] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, Logic Synthesis for Programmable Gate Arrays, Proc. Design Automation Conference, pp. 620-625, 1990.

[6] S. Chang, M. Marek-Sadowaska and T. Hwang, Technology Mapping for TLU FPGA s Based on Decomposition of Binary Decision Diagrams, IEEE Trans. Computer-Aided Design, vol. 15, pp. 1226-1236, Oct.,1997.

[7] M. Chiodo, L. Lavagno, H. Hsieh, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich, Synthesis of Software Programs for Embedded Control Applications, Proc. Design Automation Conference, pp. 587-592, 1995.

[8] M. Sauerhoff and I. Wegener, On the Complexity of Minimizing the OBDD Size for Incompletely Specified Functions, IEEE Trans. Computer-Aided Design, vol. 15, pp. 1435-1437, Nov. 1996.

[9] A. L. Oliveira, L. Carloni, T. Villa and A. Sangiovanni-Vincentelli, Exact Minimization of Boolean Decision Diagrams Using Implicit Techniques, Technical Report UCB ERL M96/16, University of California, 1996.

[10] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton, Heuristic Minimization of BDDs Using Dont Cares, Proc. Design Automation Conference, pp. 225-231, 1994.

[11] S. Chang, D. I. Cheng and M. Marek-Sadowska, Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions, Proc. the European Design and Test Conference, pp. 620-624, 1994.

[12] O. Coudert, C. Berthet and J. C. Madre, Verification of Synchronous Sequential Machines Based on Symbolic Execution, Automatic Verification Methods for Finite State systems, Springer-Verlag, pp. 365-373, 1989.

[13] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton and A. Sangiovanni-Vincentelli, Implicit State Enumeration of Finite State Machines using BDDs, Proc. International Conference on Computer-Aided Design, pp. 130-133, 1990.

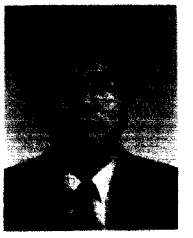
[14] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, Safe BDD Minimization Using Dont Cares, Proc. Design Automation Conference, pp. 208-213, 1997.

[15] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, and A. Sangiovanni-Vincentelli, SIS: Sequential Circuit Design Using Synthesis and Optimization, Proc. ICCD, pp. 328-333, Oct, 1992.

## 저 자 소 개

洪 裕 杓(正會員)

1991년 연세대학교 전기공학과 학사. 1993년 University of Southenr California 전기공학과 석사. 1998년 University of Southern California 컴퓨터공학과 박사. 1998년 7월-1999년 2월 Synopsys, Hillsboro에서 정형검증알고리즘 연구. 1999년 2월-현재 동국대학교 전자공학과 재직



朴 台 根(正會員)

1985년 2월 연세대학교 전자공학과 학사. 1993년 Syracuse Univ. Computer 공학석사. 1993년 Syracuse Univ. Computer 공학박사. 1991년 9월-1992년 3월 Coherent Research Inc. VLSI design engineer. 1994년 2월-1998년 2월 현대전자 System IC 연구소 책임 연구원. 1998년 3월-현재 카톨릭 대학교 컴퓨터 전자공학부 조교수. 주관심 분야는 VLSI, CAD, 병렬처리, associative processing등임