論 文
35~11~2

논리 프로그래밍을 사용한 상태도의 오류검출과 상태축소에 관한 연구

A Study on Error Check and State Reduction of State Diagram Using Logic Programming

李 克*·金 民煥*·黃 熙隆**
(Geuk Lee·Min-Hwan Kim·Hee-Yeung Hwang)

Abstract

This paper is concerned with the techniques of error check and reduction of state diagram using logic programming.

Error check program aims to check not only syntax errors but also semantic errors. And reduction program optimizes the state diagram by finding the redundant equivalence states and removig those from the set of states. The input of both program is state diagram represented as state table form. The output of error check program is error comment. The output of reduction program is equivalence reduced state table.

Both programs are implemented using Prolog. Prolog has very powerful pattern matching, and its automatic back-tracking capabilities facilitate easy-to-write error check and reduction programs.

1. 서 론

지난 수십년간 하드웨어 시스템 설계는 급속한 전자 기술의 발달에 따라 점차 복잡해져 왔으며, 이같은 복잡성이 증가함으로써 하드웨어 시스템 설계를 사람의 손으로 할 경우 설계를 위한 시간과 계발 비용이 엄청나게 커지게 되었고, 특히 오류가 발생하였을 경우 오류수정(debugging)을 하는 일이 거의 한계점에 이르게되었다. 종래의 설계법으로 마이크로 프로세서를 설계할 경우 설계에 60명이 1년 (60 man-years)이상, 오류수정에 60명이 1년 이상의 시간이 소요될 것으로 추정된다.[1]

이러한 어려움을 해결하기 위한 하나의 방법으로 CAD(Computer Aided Design) 시스템이 도입되면서 이 분야에 많은 발전이 이루어져서 현재는 설계 자동화(design automation)의 단계로 나아가고 있다. 설계 자동화란 어러가지 결과를 얻기 위하여 설계 과정에 컴퓨터를 이용하는 것을 말하며 이말은 CAD(Computer Aided Design), CAE(Computer Aided Engineering), CAT(Computer Aided Testing), CAM(Computer Aided Manufacturing)과 관련되어 주로 사용되고 있다. 보통 설계 자동화란 아날로그, 디지탈 전자시스템으로 부터 기계적인 시스템(mechanical system)에 이르기까지 광범위한 적용 분야를 포괄하고 있으며, 컴퓨터 설계분야에서는 특히 디지탈 전자 시스템에 관한 산물(product)을 얻기 위한 설계지원을 말하며 특히 VLSI설계 분야에서 많이 쓰이고 있다.[2]

본 논문의 상태도 처리 프로그램은 동기식 순차회로(synchronous sequential circuit) 설계 자동화 시스템의 일부로서 기존의 동기식 순차회로 합성 시스템[3]의 전단에 붙여서 사용하기 위해 개발되었다.

*正 會 員 : 서울大 大學院 電子計算機工學科 博士課程
**正 會 員 : 서울大 工大 電子計算機工學科 敎授 · 工博
接受日字 : 1986年 4月 29日

상태도 처리 프로그램은 상태도를 입력으로 받아 그 상태도에서 발생할 수 있는 몇가지 오류를 검출한 후, 수정된 상태도에서 동치관계의 상태들을 찾아 축소시키므로써 불필요한 이진 기억장치(binary storage device)들의 사용을 없애주도록 시도하였다.

상태도 처리 프로그램은 Prolog로 구현하였다. 최근 인공지능 언어로 각광받고 있는 Prolog는 유니피케이션(unifiction)을 통한 패턴매칭(pattern matching) 기능으로 상태도 처리와 같은 비수치 연산에 적합하며, 해를 발견하기 위한 자동백트랙킹(automatic backtracking)으로 1개 이상의 다른 해를 탐색하는 기능이 있어 프로그램 작성이 용이하다.[4]

본 논문은 크게 오류를 검출하는 부분과 상태를 축소하는 두 부분으로 구성되어 있으며 먼저 용어들을 정의하고 프로그램 활용 알고리즘을 제시한 후, 예를 들어 보이겠다.

2. 오류검출

보통의 합성 시스템에 있어서 오류검출은 크게 두 종류로 분류할 수 있다. 첫째는 모의 실험(simulation)에 의한 방법[5]으로 합성된 결과와 합성되기전의 설계 사양(design spesification)을 각각 모의 실험을 하여 같은 결과가 나오는지를 비교하는 방법이다. 이러한 방법에 있어서는 보통 어떤 검사 패턴(test pattern)을 생성하는가(generate)가 문제시된다. 두번째 방법은 형식 검증(formal verification)에 의한 방법[6]으로서 형식적인 증명에 의해 설계 결과가 제안된 사양과 맞는지를 검증한다. 일반적인 합성의 단계는 그림 1과 같다.[7] 이들 각 합성 단계에 있어서 오류는 여러 단계를 거친 단계 즉 합성의 아래 단계에서 검출될 수록 오류 수정을 위해 드는 노력은 커진다.[8] 다시 말해, 초기 단계에서 오류를 검출할수록 오류검출로 인한 비용 절감의 효과는 커진다.

본 논문에서는 합성하기에 앞서, 설계 사양이라 할 수 있는 상태도에서 오류를 검출하므로써 합성 후의 오류 수정에 필요한 노력과 비용을 절감시켜 주도록 한다. 먼저 상태도와 구문오류를 다음과 같이 정의한다.

2.1 상태도와 구문오류의 정의

오류검출을 위해 우선 상태도와 구문오류를 정의한다. 상태도는 DFA(Deterministic Finite Automata)와 유사한 아래 정의와 같은 형태를 취할 것이며 이러한 상태도의 특성을 사용하여 구문오류를 검출해낸다.

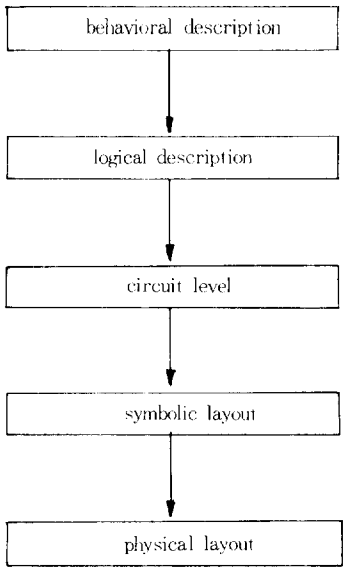

그림 1. 합성의 단계
Fig. 1. Synthesis step.

2.1.1 상태도 정의

상태도는 아래와 같이 정의한다.

〈정의 1〉 상태도

상태도 = (Q, ∑, △, δ, λ, I, F)

단 Q : 유한 갯수의 상태들의 집합

∑ : 유한 갯수의 입력 알파벳 집합

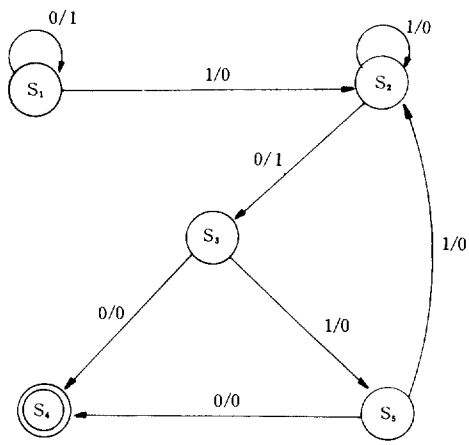

그림 2. 상태도 1
Fig. 2. State diagram 1.

△ : 유한 갯수의 출력 알파벳 집합
δ : Q×∑에서 Q로 사상시키는 다음 상태 함수
λ : Q×∑에서 △로 사상시키는 출력함수
〈여기서 ×는 Cartesian product〉
I : 초기상태(initial state)들의 집합(I⊆Q)
F : 끝상태(final state)들의 집합(F⊆Q)

〈예 1〉 그림 2와 같은 상태도는 다음과 같이 표시된다.

Q : {S1, S2, S3, S4, S5}
∑ : {0, 1}
△ : {0, 1}
δ : δ(S1, 1)=S2, δ(S1, 0)=S1, ⋯ δ(S5, 0)=S4
λ : λ(S1, 1)=0, λ(S1, 0)=1, ⋯, λ(S5, 0)=0
I : {S1}
F : {S4}

2.1.2 구문오류의 정의

정의 1과 같이 정의된 상태도에서 구문오류는 여러가지가 있을 수 있으나 본 논문에서는 설계자들이 설계시 범하기 쉬운, 다음 정의 2에서 5까지의 4가지 치명적인 오류만을 검출하는 것으로 제한한다. 각 구문오류는 다음과 같이 정의한다.

〈정의 2〉 불분명한 상태(non-deterministic state)

아래 조건을 만족하는 임의의 상태 q_i가 존재할 경우 이 q_i를 불분명한 상태라 한다.

$\neg\forall q_i,\ \exists q_j, q_k \Rightarrow \{\delta(q_i, a)=\{q_j, q_k\},$
$q_i \neq q_j,\ q_i,\ q_j,\ q_k \in Q,\ a \in \sum\}$

(예 2) 그림 3의 (a)와 같이 어떤 임의의 상태 q_i에서 1개의 입력스트링 a에 대해 2개 이상의 상태로 천이하는 경우이다. 이러한 경우는 시스템의 상태에 따라 상태천이(state transition)가 불규칙적으로 일어나므로 시스템의 동작을 예측할 수 없게 된다.

〈정의 3〉 성공적이지 못한 종결(unsuccessful finish)

아래 조건을 만족하는 q_i가 존재할 경우 성공적이지 못한 종결이라 한다.

$\neg\forall q_i, a, \nexists q_j \Rightarrow \{\delta(q_i, a)=\{q_j\},$
$q_i \notin F,\ q_i \in Q,\ a \in \sum\}$

(예 3) 이러한 상태는 그림 3의 (b)와 같이 더이상 상태천이가 일어날 수 없는 상태인 q_i까지 도달했으나 이 상태 q_i가 끝상태들 중의 하나가 아닌 경우이다. 즉 끝상태가 아닌 상태에서 더 이상 진행

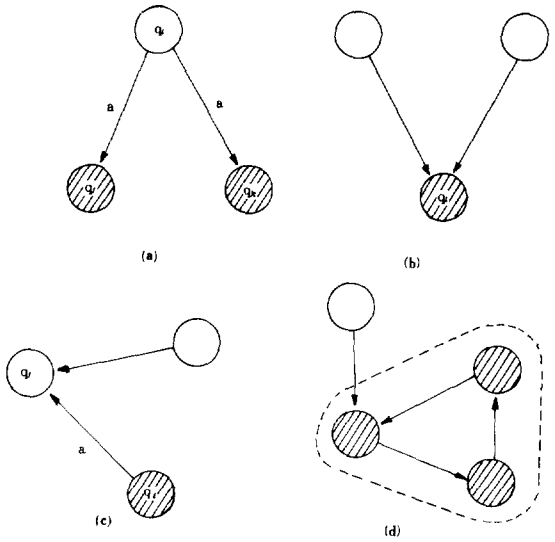

그림3. 구문오류의 종류
(a) 불분명한 상태 (b) 성공적이지 못한 종결
(c) 도달 불가능 상태 (d) 순환 조건

Fig. 3. Syntax arrors.
(a) non-deterministic state
(b) unsuccessful finish
(c) unreachable state
(d) loop condition

할 수 없는 경우이다.

〈정의 4〉 도달 불가능 상태(unreachable state)

아래 조건을 만족하는 q_i가 존재할 경우, 이 q_i를 도달 불가능 상태라 한다.

$\neg\forall q_i, a\ \exists q_j \Rightarrow \{\delta(q_j, a)=\{q_i\},$
$q_i \notin I,\ q_i \in Q,\ a \in \sum\}$

(예 4) 그림 3의 (c)와 같이 상태 q_i에서 나가는 가지(outgoing edge)들만 존재하고 들어오는 가지(incomming edge)는 존재하지않는 경우이다. 이러한 상태는 초기상태들에만 존재해야 하며, 그렇지 않은 경우 오류로 검출해야 한다.

〈정의 5〉 순환 조건(loop conditipon)

임의의 상태에서 어떤 단일한 경로를 거쳐 다시 그 상태에 도달하는 경우를 순환조건이라 한다.

(예 5) 그림 3의 (d)에서의 점선으로 묶여진 빗금친 상태들의 경우로서, 이 경우는 설계자가 고의로 이와 같은 상태를 만드는 경우가 있을 수 있으므로 단지 '주의'라는 문장만을 출력하고 뒤에서

설명할 상태축소(state reduction) 프로그램으로 넘어가 상태축소 프로그램에서 이 부분을 축소해준다.

2.2 Prolog를 이용한 구문오류의 검출

본 절에서는 2.1.1절에서 기술한 정의에 따라 상태도를 Prolog로 표현하고 오류를 검출하는 방법을 설명한다.

2.2.1 Prolog에 의한 상태도의 표현 방법

그림 4와 같은 상태도가 있을 경우 상태 a와 b에서 입력 스트링은 v와 x이고 출력 스트링은 w와 y이다. 이런 상태도의 표현은 설계자에 따라 여러가지가 있을 수 있겠으나, 본 논문에서는 다음과 같이 표현했다.

① sd([v, w], a, a).
② sd([x, y], a, b).

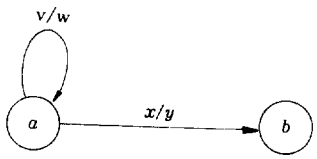

그림4. 상태도 2
Fig. 4. State diagram 2.

①은 상태 a에서 a로 천이하는 가지를 나타내는 부분으로서 입력스트링이 v일때 출력스트링 w를 생성해 줌을 나타낸다. 역시 ②도 같은 방법으로서 입력이 x일때 y를 생성해 주며 a에서 b로 천이하는 가지를 나타내고 있다.

이 표현법에 의해 그림 2의 상태도를 Prolog로 나타낸 것은 다음과 같다. 이것들은 입력 화일의 형태로 Prolog에 프로그램과 함께 입력되어 데이타 베이스의 역활을 하게된다.

```
sd([0, 1], 1, 1).
sd([1, 0], 1, 2).
sd([1, 0], 2, 2).
sd([0, 1], 2, 3).
sd([0, 0], 3, 4).
sd([1, 0], 3, 5).
sd([1, 0], 5, 2).
sd([0, 0], 5, 4).
```

2.2.2 Prolog를 이용한 구문오류의 검출 방법

오류검출 프로그램 error-check는 manual과 automatic 두 부분으로 이루어져 있다.

manual은 사용자가 프로그램의 질의어(query)에 입력스트링과 초기상태를 입력해주면 프로그램은 그 입력스트링에 대응되는 출력스트링과 상태의 흐름을 출력해준다. 그러면 설계자가 출력된 스트링과 상태의 흐름을 보고 예상했던 결과와 다르게 동작하는지를 점검해보아야 한다. 이 manual프로그램은 단지 다음과 같은 4줄의 Prolog 프로그램으로 간단히 해결된다.

```
manual([], W, M, Y).
manual([H|T], A, [V|W], [X|Y]) :-
        sd([H, X], A, M),
        manual(T, M, W, T).
```

(예6) 그림 2의 상태도를 2.2.1절에서와 같이 Prolog로 표현해 입력했을 경우 질의어로서 다음과 같이 줄 수 있다.

```
? -manual([0, 1, 1, 0, 1, 0], 1, State, Output).
```

이에대해 다음과 같은 출력이 나온다.

State=1, 2, 2, 3, 5, 4
Output=1, 0, 0, 1, 0, 0

automatic 부분은 manual과 달리 질의어에서 입력스트링 따위를 줄 필요가 없으며 단지 ?-error-check라고만 해주면 된다. 그러면 프로그램 자체가 초기상태에서 끝상태에 이르는 모든 상태를 자동적으로 따라가면서 앞에서 정의한 구문오류(정의 2에서 정의 5의 상태)들을 자동적으로 검출해준다. 예를 들어 그림 3의 (a)와 같은 '불분명한 상태'를만나면 다음과 같은 오류설명(error comment)을 출력해준다.

"State q_i is non-deterministic.
Because state q_i has different transition states q_i and q_k with a input string a."

이 부분의 Prolog 표현은 다음과 같다.

```
non-determ(Current1) :-
    sd([Input1, Output1], Current1, Next1),
    sd([Input1, Output2], Current1, Next2),
    Next1\=Next2,
    write('State q is…').
```

즉 어떤 현재상태(변수 Current1)에서 나가는 가지가 둘 이상 존재하여(sd라는 atom이 두개 존재) 그 가지들의 입력 스트링(변수 Input1)은 같고, 천이된 상태들이 다르면(Next1\=Next2) 오류설명(write 문장)을 출력해 준다.

여기서 볼 수 있는 바와 같이 정의 2에서 나타난 오류 정의가 그대로 프로그램에 자연스럽게 나타나

고 있다. 이러한 이유는 Prolog가 1계 술어 논리(first order predicate logic)에 기반을 두고 있으므로[9] 정의 2의 논리적 표현과 잘 일치되고 있는 것이며, 이로 인해 상태도 처리와 같이 일정한 정의에 따라 각 상태를 점검하는 프로그램은 Prolog를 사용하므로써 잇점이 있다. 이러한 점들은 다음과 같이 Prolog 규칙(rule)의 형태로 표현된 '성공적이지 못한 종결'과 '도달 불가능 상태'에서도 잘 나타난다. '성공적이지 못한 종결'은 다음과 같다.

"Your state diagram can't be finished successufully. Because state q_i is not final state."

```
un-succfini(Current1) : -
    sd([Input, Output], Current1, Next).
un-succfini(Current1) : -
    final(States),
    member(Current1, States).
un-succfini(Current1) : -
    write('Your state…').
```

'도달 불가능 상태'는 다음과 같다.

"State q_i is unreachable.
Because q_i has only out-going edges."

```
un-reachable(Current1) : -
    sd([Input, Output], Anystate, Current1).
un-reachable(Current1) : -
    initial(States),
    member(Current1, States).
un-reachable(Current1) : -
    write('State q is…').
```

모든 상태들을 방문하는 일은 Prolog의 기능중의 하나인 백트랙킹을 이용해 초기상태에서 끝상태 까지의 경로를 따라가도록 하여 각 상태가 구문오류에 정의된 상태중의 하나인가를 점검하게 하면 어렵지 않게 해결된다. 이때 물론 입력 데이타 베이스에는 초기상태와 끝상태를 initial[1]과 final[4]와 같이 미리 써넣어 주어야 한다.

2.3 Prolog를 이용한 의미오류(semantic error)의 검출

의미오류(semantic error)의 경우는 지금까지 기술한 구문오류와 달리 상태도의 표현상의 오류가 아닌 설계상의 논리적 오류이며, 각 문제에 종속된(problem dependent) 경우이므로 이를 검출할 수 있는 통상적인 방법은 존재하지 않는다. 상태도에서의 의미오류 검출을 위해 가장 쉽게 생각할 수 있는 일반적인 방법은 가능한 모든 입력스트링에 대해 이에 따른 모든 출력스트링과 상태의 흐름을 출력해 주어 설계자가 의미오류를 찾을 수 있도록 도와주는 방법이 있을 수 있으나, 이는 대부분의 상태도의 경우 그들이 갖는 특성 때문에 비다항식(Non-Polinomial, 이후 NP로 약술함) 문제가 발생하므로 실제 구현은 어렵다. 다른 한가지는 상태도를 적당히 잘라서 상태의 천이에 따른 입력 스트링과 출력 스트링을 정규 수식(regular exprsession)의[10] 형태로 출력해주는 방법이 있으나, 이 방법 역시 정규수식 형태의 출력이 판독성(readability)에 있어서 상태도보다 의미오류 검출에 효용성이 있는지는 의문시된다.

이에 본 논문에서는 주어진 문제에서 반드시 지켜야할 원칙이나 발생 가능한 의미 오류들을 Prolog의 규칙(rule)의 형태로 표현하여 프로그램 수행 단계에서 오류를 검사하는 방안을 제시한다. 즉 주어진 문제에 대한 지식을 이용하여 오류를 검출해 주는 방법이다. Prolog 프로그램은 각기 규칙의 형태로 구성되어 있으므로 규칙의 삽입이 용이하며 또한 그 위치 선택도 임의로 결정할 수 있으므로프로그램 구현이 용이하다. 예를 들어 150원을 넣으면 커피를 내주는 커피 자동 판매기를 설계할 경우 설계자가 그림 5와 같이 실수로 틀리게 설계했다고 가정해보자. 문제를 간단히하기 위해 동전은 100원과 50원 동전만 사용하고 거스름돈은 고려하지 않는다.

이 커피 자동 판매기의 경우 출력이 1이면 커피를 내주도록 설계한 것인데, 쉽게 알 수 있듯이 앞에서 정의된 구문오류는 전혀 없다. 하지만 상태

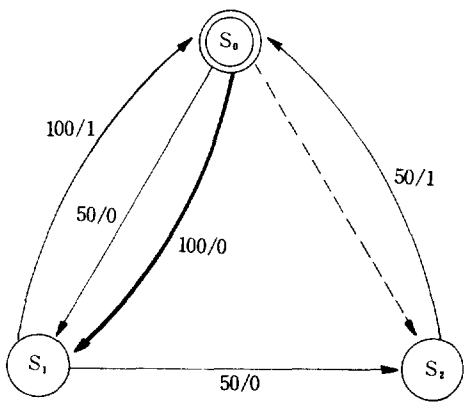

그림5. 커피 자동 판매기의 상태도
Fig.5. State diagrame of coffee machine.

S0에서 S1으로 가는 굵은선은 S0에서 S2로 가는 점선의 위치로 가야하는 것을 잘못 설계한 것이다. 처음 50원 동전을 넣으면 150원 투입시 커피가 나오지만, 100원 동전을 넣으면 200원을 넣어야 커피가 나오도록 잘못 설계되어 있다. 이와 같은 상태도에서 만일 초기상태에서 끝상태에 이르는 입력스트링의 합이 150이 되어야 한다는 규칙을 첨가해주면 간단히 오류를 검출해낼 수 있다. 이에대한 Prolog 규칙은 다음과 같다.

check-sum(X) : −total(Y), X=Y.
check-sum(X) : −write('Sum'), write(X),
write('is incorrect.').

즉, 초기상태에서 끝상태에 이르는 입력 스트링의 합이 총계(total(X))와 다르면 오류임을 출력해 준다.

3. 상태축소

상태도의 오류 검출이 끝나면 이의 논리도 (logic diagram) 실현 단계 이전에 상태축소 과정이 필요하다. 이 과정 역시 Prolog를 사용해 시도해 보았다.

상태축소 프로그램은 상태도 내에 동치관계(equivalence)의 상태들이 있는지 검토해 축소시켜 준다. n을 이진기억장치(flip flop따위)의 최소수, v를 상태의 수라고 할 때 n과 v는 다음과 같은 관계가 있다.

$2^{n-1}<v\leq 2^n$

이 관계에서 알 수 있는 바와 같이 상태수의 축소는 전체 기억장치(storage device)의 수를 감소시켜 하드웨어 비용을 절감시키는 효과를 가지고있다. 먼저 상태축소에 필요한 용어와 알고리즘을 설명한 후 예를 들어 보이겠다. 본 절에서 사용하는 용어와 축소절차는 Torng[12]의 방법을 따랐으므로 일체의 증명은 생략하고 Prolog의 표현법을 첨가해 구현한 부분만을 중심적으로 설명하기로 한다.

3.1 상태축소를 위한 정의와 알고리즘

3.1.1 상태축소를 위한 정의

상태축소를 위해 동치관계 상태와 레벨 k 에서의 k−동치관계를 다음과 같이 정의한다.

〈정의 6〉 동치관계 상태(equivalence state)

상태 q_i와 q_j가 다음 관계를 만족할때 q_i와 q_j는 동치관계에 있다고 한다.

$\forall T,\ \exists q_i, q_j \ni \{\delta(q_i, T)=\delta(q_j, T),\ q_i, q_j\in Q,$
T is an input string$\}$

이 정의에 레벨 k에서의 동치관계 클래스들을 묶어내기 위한 k−동치관계는 아래와 같이 정의한다.

〈정의 7〉 k−동치관계 (k-equivalence)

상태 S_i와 S_j가 다음 관계에 있을때 S_i와 S_j는 k−동치관계에 있다고 한다.

$\exists S_i, S_j, \forall T$ of length$\leq k \ni \{\lambda(S_i, T)=\lambda(S_j, T),$
T is an input string$\}$

3.1.2 상태축소를 위한 알고리즘과 정리

레벨 k에서의 k−동치관계 클래스들의 집합을 P_k로 표시하면 상태축소 알고리즘은 다음과 같다.

단계 1>k=1로 놓는다.

단계 2>k−동치관계 클래스들과 P_k를 찾는다.

단계 3>$P_k\neq P_{k-1}$이면 k를 증가시킨 후 단계2로, $P_k=P_{k-1}$이면 끝낸다. (단 $P_0=\{\phi\}$)

이 축소 알고리즘을 상황에 따라 좀 더 간략화 시킬 수 있는 정리들로서 다음과 같은 것이 있다.

정리 1>만일 P1이 단지 1개의 원소만 가지면 P2=P1이다. 즉 각 상태들의 출력이 모두 같으면 모든 상태들은 1개의 상태로 축소될 수 있다.

정리 2>u를 초기상태의 수라고 가정할 경우 u>2이면 $P_u=P_{u-1}$이다. 즉 초기상태의 수보다 하나 적은 수 만큼의 P만 찾으면 된다.

3.2 상태축소의 예

그림 6 과 같은 상태도가 있을 경우 상태도를 표시하는 Prolog의 입력 데이타 베이스 초기값은 오류검출 프로그램에서와 같이 다음과 같이 표현된다. 오류검출 프로그램과 같은 표현법을 쓴 이유는 오류검출이 끝난 상태도를 바로 상태축소 프로그램의 입력으로 받아 축소해주기 위함이다.

```
sd([0, 0], 0, 0).
sd([1, 0], 0, 1).
sd([1, 0], 1, 3).
sd([0, 0], 1, 2).
sd([1, 0], 2, 3).
sd([0, 0], 2, 0).
sd([0, 0], 4, 0).
sd([1, 1], 4, 5).
sd([0, 0], 3, 4).
sd([1, 1], 3, 5).
sd([0, 0], 6, 0).
sd([1, 1], 6, 5).
sd([1, 1], 5, 5).
sd([0, 0], 5, 6).
```

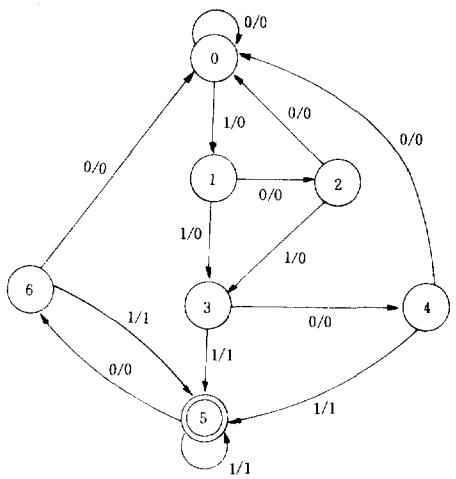

그림6. 축소되기 전의 상태도.
Fig. 6. State diagram before the reduction.

일단 받아들여진 상태도는 동치관계 클래스를 찾기 이전에, 이들 동치관계 클래스를 쉽게 찾을수 있도록 일시적으로 형태를 변형한다. 그림 6의 상태도는 sdt라는 술어(predicate)를 가진 입력 알파벳에 대해 오름차순으로 나열된 형태의 중간 데이타 베이스가 형성된다.

중간 데이타 베이스가 만들어지면 1-동치 관계 클래스를 찾는다. 1-동치관계 클래스는 정의 7에 따라 단지 임의의 입력에 대해 동일한 출력을 생성하는 상태들을 찾으면 된다.

이들은 eq-class란 술어명(predicate name)을 가진 다음의 두개의 집합으로 기억된다.

eq-class(1, 1, [0, 1, 2]).
eq-class(1, 2, [3, 4, 5, 6]).

첫번째 원소 '1'은 1-동치관계임을 나타내며, 다음 원소는 1-동치관계 집합의 번호로써 클래스 {0, 1, 2}와 {3, 4, 5, 6}을 구분하기 위해 각각 붙여졌다. 이 1-동치관계를 찾는 부분은 실제 프로그램내에는 oneeq라는 술어명으로 들어있다.

다음은 2-동치관계 이상을 찾아내는 작업으로써 현재의 동치관계 클래스에서 각 집합의 원소들이 각 입력에 대해 같은 상태로 전이하는가를 검토해야 한다. 다시 말해 같은 동치관계의 집합들로 전이하면 그대로 두고 그렇지 않으면 분리한다. 예를 들어 그림 6에서 도출된 위의 1-동치관계의 첫번째 집합에서 '1'과 '2'의 상태는 입력이 '0' 일 때 '2'와 '0'의 첫번째 집합으로 전이하고 다음과 같다.

eq-class(2, 1, [0]).
eq-class(2, 2, [1, 2]).
eq-class(2, 3, [3, 5]).
eq-class(2, 4, [4, 6]).

이러한 분리 작업을 n-동치관계 클래스가 (n-1)-동치관계 클래스와 같을 때까지 계속한다. n-동치관계 클래스가 (n-1)-동치관계 클래스와 같으면 더 이상의 동치관계의 클래스는 없으므로 동치관계에 있는 상태들을 같은 상태로 축소해준다. 이 예에서 상태도는 아래와 같이 4-동치관계클래스 까지를 생성해 보아야 동치관계 클래스가 더 이상 없음을 알 수 있다.

eq-class(3, 1, [0]).
eq-class(3, 2, [1]).
eq-class(3, 3, [2]).
eq-class(3, 4, [3, 5]).
eq-class(3, 5, [4, 6]).
eq-class(4, 1, [0]).
eq-class(4, 2, [1]).
eq-class(4, 3, [2]).
eq-class(4, 4, [3, 5]).
eq-class(4, 5, [4, 6]).

위에서 3과 5, 그리고 4와 6은 같은 상태임을 알 수 있으므로 5를 3에 4를 6에 포함시켜 축소된 상태표를 출력시킨다. 이 상태도는 그림 7과 같다.

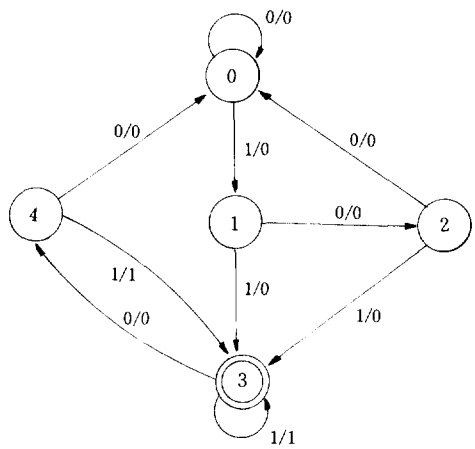

그림7. 축소된 상태도
Fig. 7. Reduced state diagram.

3. 결 론

본 논문에서는 오류검출 프로그램을 Prolog를이

용해 구현했으며 구문오류뿐만 아니라 의미오류도 검출할 수 있는 방법을 제시하고 있다. 앞에서 기술한 바와 같이 모든 경로(edge)를 따라가며 오류를 찾는 문제는 Prolog의 백트랙킹으로 해결되며, 문제에 종속된 규칙을 첨가해주므로써 의미오류를 검출할 수 있는 방안을 제시하고 있다. Prolog는 프로그램이 각기 규칙의 형태로 구성되어 있어 프로그램을 확장 시키거나 다른 Prolog 프로그램에 연결시키기가 편리하다는 잇점이 있다. 오류검출 프로그램은 오류검출을 상대도의 수준에서 해주므로써 기존의 논리도에서의 오류검출보다 비용 절감의 효과가 크며, 상태축소 프로그램은 동치관계 상태들을 찾아 축소해주므로써 플립플롭과 같은 기억장치의 사용을 최소화 시켜준다.

본 논문에서는 의미오류의 검출에 있어서 Prolog 규칙을 이용하므로 상태도의 설계자는 Prolog를 어느 정도 이해하여야 한다. 이런 점을 보완하기 위해 앞으로의 연구 방향은 기존 시스템을 확장하여 사용자가 구문오류뿐만 아니라 의미오류까지도 쉽게 정의하여 사용할 수 있는 전문가 시스템(expert system)의 구축이 필요하다 하겠다. 또한 상태축소는 완전 서술(completely specified) 상태도에 대해서만 적용되므로 불완전 서술(incompletely specified) 상태도에도 적용시킬 수 있도록 확장한다면 상태축소의 효과는 더욱 커질 것이다.

참 고 문 헌

1) M. Feuer, "VLSI Design Automation: An Introduction", Proc. IEEE, Vol. 71, January 1983.
2) H. Ofek, S.Lisco, "Design Automation", IEEE Design & Test, Vol. 1, Fevrury 1984.
3) 김 경식, 황 희용, "컴퓨터를 이용한 동기식 순차논리회로의 설계", 서울대학교 석사 학위 논문, 1984.
4) E. A. Torrero et al., Next Generation, IEEE Spectrum, Vol. 20, November 1983.
5) K. Tham, R. Willoner, D. Wimp, "Functional Design Verification by Multi-Level Simulation", in Proc. 21st Design Automation Conf., June 1984.
6) A. C. Parker, F. Kurdahl, M.Mlinar, "General Methodology for Synthesis and Verification of Register Transfer Design", in Proc. 21st Design Automation Conf., June 1984.
7) A. S. Wojick, J. Kljaich, N. Srinivas, "A Formal Design Verification System Based on an Automated Reasoning System", in Proc. 21st Design Automation Conf., June 1984.
8) B. W, Böhem, Software Engineering Economics, Prentice Hall, 1984.
9) P. Gray, Logic. Algebra and Databases, John-Wiley & Sons, 1984.
10) J. E. Hopcropt, J. D. Ullman, Introduction to Automata Theory Languages and Computation, Addison-Wesley, 1979.
11) M. Fujita, H. Tanaka, T. Moto-oka, "Specifing Hardware in Temporal Logic and Efficient Synthesis of State Diagram Using Prolog," in Proc. 5th Gen. Comp. Sys. Conf., November 1984.
12) H. C. Torng, Switching Circuit; Theory and Logic Design, Addison-Wesley, 1972.
13) W. F. Clocksin, C. S. Mellish, Programming in Prolog. Springer Verlag, 1984.
14) F. Maruyama, M. Fujita, "Hardware Varification", IEEE Computer, Vol. 18, February 1985.
15) F. Maruyama, T. Mano, K. Hayashi, T. Kakuda, N. Kawato, T. Uehara, "Prolog Based Expert System for Logic Design", in Proc. 5th Gen. Comp. Sys., November 1984.
16) A. S. Wojcik, "Formal Design Verification of Digital Systems", IEEE Trans. on Computer, Vol. 32, September 1983.
17) L. Wos, R. Overbeak, E. Lusk, J. Boyle, Automated Reasoning: Introduction and Applications, Prentice Hall, 1984.