하계학술회의
논문85-8-2

# Prolog를 이용한 State Diagrame의 처리

## State Diagram Manegement Using Prolog

이 극*
조 동섭
황 희웅

서 울 대 학 교
전자계산기공학과

<요 약>

본 논문은 특정한 하드웨어를 구성함에 앞서 그 시스템이 State-diagram으로 표현된 것을 입력으로 받아 state-diagram이 맞는지를 verify하는 방법을 제시하며 equivalence state를 찾아 reduce해준다.
Program의 실현은 Prolog로 하였다.

## I. 서 론

하드웨어나 소프트웨어 시스템을 구성하는데 있어서 그 시스템이 정확히 작동하는지 점검하는 일은 매우 중요하다. 소프트웨어 시스템의 경우 이를 위해 많은 연구가 이루어지고 있으며 근자에 어느정도 성과를 이루고 있다.[1]

하드웨어 시스템의 경우에 있어서 어떤 시스템을 실제 하드웨어로 구성하기 전에 설계한 시스템이 바르게 작동하는지 점검하는 일은 시스템의 설계비용을 절감시키는데 결정적인 역할을 하며 이를 위해 여러가지 방법들이 연구되고 있다.[2,3]
이에 본 논문은 하드웨어를 구성하기 위한 design specification이라 할 수 있는 state-diagram(이후 s-d이라 약술함)을 입력으로 받아 그 s-d으로 하드웨어를 구성할 경우 바르게 작동 하는지를 verify하는 방법을 제시하며 이를 prolog의 강력한 pattern match기능과 automatic backtracking기능을 최대한 이용해 구현하므로서 syntax error뿐 아니라 semantic error까지 check할수 있는 가능성을 제시하며 또한 s-d에서 equivalence state들을 찾아 reduce해줌으로서 redundante한 storage device들의 사용을 없애준다.

## II. 본 론

s-d은 DFA(Deterministic Finite Automata)와 유사한 형태를 취하고 있으며 이 s-d의 특성에 의해 syntaxError를 검출해 낼 수 있다.

우선 s-d을 정의하면 다음과 같다.

<정 의1> s-d = $(Q,\Sigma,\Delta,\delta,\lambda,I,F)$

$Q$: finite set of states

$\Sigma$: finite input alphabet

$\Delta$: finite output alphabet

$\delta$: next state function mapping from $Q\times\Sigma$ to $Q$

$\lambda$: output function mapping from $Q\times\Sigma$ to $\Delta$

$I$: a set of initial states ($I \subseteq Q$)

$F$: a set of final states ($F \subseteq Q$)

이 정의에 따라 s-d의 주요한 syntax error는 다음과 같이 정의한다.

<정 의2> 다음 경우를 ambiguous 라 한다.

$$\neg\forall q_i, \exists q_j, q_k \Rightarrow \{\delta(q_i, a_i) = \{q_j, q_k\} \mid q_i, q_j, q_k \in Q,\ a_i \in \Sigma$$

<정 의3> 다음 경우를 unsuccessful finish라 한다.

$$\neg\forall q_i, a_i, \exists q_j \Rightarrow \{\delta(q_i, a_i) = \{q_j\} \mid q_i \notin F, q_i \in Q\}$$

<정 의4> 다음 경우를 un-reachable state라 한다.

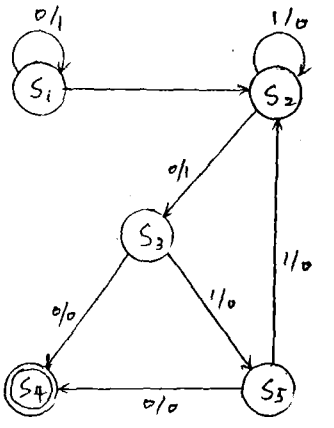
$$\neg\forall q_i, a_i, b_i, q_j, \exists q_k \Rightarrow \{\delta(q_i, a_i) = \{q_j\} \cap \delta(q_k, b_i) = \{q_i\} \mid q_i, q_j, q_k \in Q, q_i \notin F, a_i, b_i \in \Sigma\}$$

<정 의5> 임의의 state에서 어떤 단일한 path로 다시 그 state에 도달할 경우 loop condition이라 한다.

prolog program verify는 각 s-d의 모든 path를 따라가며 위의 정의 2~5에 부합하는 state를 만나면 error임을 알려준다. 모든 path를 따라가는 방법은 prolog의 automatic backtracking으로 간단히 처리된다.

semantic error의 경우는 s-d자체의 오류가 아닌 s-d design시의 logical한 error이므로 이를 check할 수 있는 위와 같은 formal한 방법은 존재하지 않는다. 가능한 모든 input string과 이에따른 모든 state와 output string을 generate해 designer가 logical error를 찾을수 있도록 도움을 주는 방법이 있으나 이는 NP-problem이 발생하므로 실제 구현이 어렵다. 본 논문은 이와 같은 semantic error check를 위해 두가지 방법을 제시한다.

첫번째 방법은 가능한 모든 다른 path를 찾아가며 이에 따른 input string과 state, output string을 생성해 Regular Expression(이후 R.E로 약함)의 형태로 출력해 주는 방법이다. R.E form의 출력은 NP-problem의 발생없이 가능한 모든 input string을 generate한것과 같은 효과를 얻을 수 있다. 예를 들어 다음과 같은 s-d의 경우 가능한 path를 R.E의 형태로 표시한 출력은 다음과 같다.




```
path i) 1 ->2 ->3 ->4
   input string:0*1 1*0 0
  output string:1*0 0*1 0

path ii) 1 ->2 ->3 ->5 ->4
   input string:0*1 1*0 1 0
  output string:1*0 0*1 0 0

path iii) 1 ->(2 ->3 ->5)#->4
   input string:0*1 1*(0 1 1)#0
  output string:1*0 0*(1 0 0)#0

path iv) 1 ->(2 ->3 ->5)#->2 ->3 ->
         4
   input string:0*1  1*(0 1  1)#1
                     (0+1*)1 0
  output string:1*0  1*(1 0  1)#1
                     (1+0*)0 0
```

verify program은 이러한 가능한 모든 path를 prolog를 이용해 스스로 generate해준다. 그러나 이러한 방법은 designer가 string을 보고 s-d의 logical error를 직접 찾아내야 하며 R.E에 익숙해져 있어야 하는 단점이 있다.

semantic error를 check하는 또 한가지 방법은 각 problem에 oriented된 rule을 verify program에 첨가하는 방법이다. prolog program의 장점은 program이 모두 rule의 형태로 표시되므로 rule의 첨가는 어렵지 않다. 그러나 designer가 prolog에 어느정도 익숙해져야하는 단점이 있다.

verify에서 error check가 끝난 s-d은 equivalent state 들이 있는지를 check 할 수 있다. v를 s-d의 state들의 수 라 하고 n을 binary storage device (flip flop따위)의 최소수라 하면 n과 v의 관계는 다음과 같다.

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

이 관계에서 알 수 있는바와 같이 state수의 reduce는 전체 storage device 수를 감소시켜 hard ware cost를 감소시키는 효과를 가지고 있다. 먼저 equivalent state는 다음과 같이 정의한다.

<정 의6> equivalent state

$$\forall a_i, \exists g_i, g_j \rightarrow \{\lambda(g_i, a_i) = \lambda(g_j, a_i) \mid g_i, g_j \in Q,\ a_i \in \Sigma\}$$

이 정의에 의해 equivalent state를 묶어내기 위한 k-equivalece는 다음과 같이 정의한다.

<정 의7> state i와 state k가 다음 관계가 있을때 k-equivalence 라 하고 $S_i \overset{k}{\equiv} S_j$ 로 표현한다. $\forall T$ of length $\leq k \rightarrow$

$$\{\lambda(S_i, T) = \lambda(S_j, T) \mid T \text{ is input string}\}$$

각 k-equivalence class에 있어서 level k에서의 모든 equivalence class들의 집합을 $P_k$로 표현한다면 reduction procedure는 다음과 같다.

단 계1> i = 1로 놓는다.

단 계2> i-equivalence class들과 $P_i$를 찾는다.

단 계3> $P_i \neq P_{i-1}$이면 i를 증가시키고 단계1로 가며 $P_i = P_{i-1}$이면 끝낸다.

(단 $P_0 = \langle \phi \rangle$ )

이 reduction procedure를 좀 더 간단히 해 줄 수 있는 정리들은 다음과 같다.

정 리1> 만일 P1이 단지 1개의 element만 가지면 P2=P1 이다.

정 리2> u를 initial state의 수라고 가정할 경우 $u \geq 2$이면 $P_{u-1} = P_u$이다. 즉 $P_{u-1}$이상의 equivalence class를 generate할 필요가 없다.

## III. 결 론

verify program은 prolog를 이용해 구현했으며 syntex error와 semantic error를 check할 수 있는 방법을 제시하고 있다. 앞에서 기술한 바와 같이 모든 path를 찾는 문제는 prolog의 backtracking으로 해결되며, problem oriented rule을 첨가해 줌으로서 semantic error를 어느 수준까지 check해 줄 수 있는 장점이 있다.

또한 equivalence state 들을 찾아 reduce 해줌으로서 storage device 들의 사용을 minimize 시킨다. 이외에도 optimal state assignment 문제가 남아 있으며 이는 storage device가 아닌 이 storage device를 구동시키는 combinatinal logic을 minimize 시키는 부분이다. 앞으로 이를 prolog로 implement할 예정이다.

[ 참 고 문 헌 ]

1. R.L. Schwartz, P.M. Mellier-Smith, "Formal Specification and Mathmatical Verification of SIFT: A Fault Tolerant Fight Control System", SRI International, Menlo Park, California, TR CSL-133, January, 1982.

2. H.G. Barrow, "Proving the Correctness of Digital Hardware Designs", Proc. of AAAI, January, 1983.

3. M. Fujita, H. Tanaka, T. Moto-oka, "Specifying Hardware in Temporal Logic and Efficient Synthesis of State Diagram Using Prolog", Proc. of 5Gen. Comp. Sys., November, 1984.

4. J.E. Hopcroft, J.D. Ullman, Intoduction to Automata Theory Languages and Computation, Addison-Wesley, 1979.

5. H.C. Torino, Switching Circuit: Theory and Logic Design, Addison Wesley, 1972.

6. W.F. Clocksin, C.S. Mellish, Programming in Prolog, Springer Verlog, 1981.

7. F. Maruyama, M. Fujita, "Hardware Verification", IEEE Computer, February, 1985.