# 논리언어의 확장 및 병렬 추론 기계의 설계 및 구현

# The Design and Implementation of A Parallel Inference Machine and An Extension of A Logic Language

연구기관
한국과학기술원

# 제 출 문

과학기술처 장관 귀하

본 보고서를 "차세대 컴퓨터 개발" 사업의 세부과제인 "논리 언어의 확장 및 병렬 추론 기계의 설계 및 구현" 사업의 최종 보고서로 제출합니다.

1989년 7월

주관연구기관명 : 한국과학기술원
협동연구기관명 : 한국과학기술원
총괄연구책임자 : 조 정 완 (전산학과 교수)
연구 책임자 : 조 정 완 (전산학과 교수)
연 구 원 : 맹 승 렬 (전산학과 조교수)

# 여 백

# 요 약 문

I. 제 목

논리 언어의 확장 및 병렬 추론 기계의 설계 및 구현

II. 연구 개발의 목적 및 중요성

다가오는 고도의 정보화 사회에서 당면하게 될 다량의 지식 정보를 효과적으로 처리하기 위해서는 기존의 컴퓨터 시스템과는 다른 구조를 갖는 컴퓨터 시스템이 필요하다. 본 연구의 목적은 이런 다량의 지식 정보를 효율적으로 처리할 수 있는 차세대 컴퓨터 시스템(지식 정보 처리 시스템)을 개발하는데 있다. 이를 위하여 첫째로, 추론 기능을 제공하면서 인공지능 언어로서 많이 사용되고 있는 논리 언어를 고속으로 수행시킬 수 있는 병렬 추론 컴퓨터를 개발하여 논리 언어에 기초한 많은 인공지능 응용 프로그램을 고속으로 수행시킬 수 있는 환경을 조성하고, 둘째로, 다량의 지식을 효과적으로 표현하기 위한 인공 지능 시스템의 핵심인 지식 표현 시스템을 개발하고 이에 대한 프로그래밍 환경을 구축하는 것이 본 연구의 목적이다.

III. 연구 개발의 내용 및 범위

본 연구의 주요 내용으로 추론 컴퓨터 분야에서는, 1-2차년 동안 연구해온 XWAM-I과 XWAM-II를 결합하여 논리 언어에 기초한 새로운 병렬 추론 기계 X-WAM을 설계하고 그 성능을 평가한다. 또한 이런 병렬 추론 기계의 시제품을 설계하고 구현한다. 지식 표현 분야에서는 그 동안 본 연구에서 개발해온 혼성 지식 표현 시스템인 Sphinx를 완성하고, 지식 표현 시스템의 프로그래밍을 위한 환경을 조성한다. 이 두가지 연구 이외에 논리 언어의 표현력을 확장하기 위한 함수 논리 언어 시스템의 구현 및 환경 조성에 관한 연구를 수행한다.

IV. 연구 결과 및 활용에 관한 건의

당 해년도의 연구 결과는 다음과 같다.

- 병렬 추론 기계에 대한 수행 모델의 개선 및 성능 평가

  논리 언어의 AND 병렬성을 추구하는 XWAM-I과 OR 병렬성을 추구하는 XWAM-II를 결합한 X-WAM을 설계하였으며, 시뮬레이션을 통하여 그 성능을 평가하였다.

- X-WAM 다중 처리기 시스템 개발
  병렬 추론 기계의 시제품으로서 386 PC를 호스트로 하고 X-WAM을 서버로 하는 전체 시스템을 완성하였으며, 윈도우에 바탕을 둔 사용자 인터페이스를 개발하였다.
- Prolog 전용 프로세서의 설계
  Prolog 수행 시간의 대부분을 차지하는 단일화 연산의 수행 속도를 개선하기 위한 하드웨어를 첨가한 Prolog 전용 프로세서를 설계하고 그 성능을 평가하였다.
- 함수 논리 언어의 개발 및 구현
  논리 언어의 표현력을 확장시키는 연구로 함수 논리 언어 일종인 Aflog에 대한 추상기계를 설계하고 시뮬레이터를 완성하였다.
- 혼성 지식 표현 시스템의 완성
  Sun 워크스테이션 상에서 윈도우를 기반으로하는 혼성 지식 표현 시스템 Sphinx를 완성 하였다.
- 혼성 지식 표현 시스템의 프로그래밍 환경 구축
  Sphinx의 지식베이스인 ABox와 TBox를 보다 편리하게 구축하고 유지하기 위하여 지식베이스 편집기와 지식베이스 브라우저를 구축하였다.

이런 연구 결과의 활용 방안으로는 혼성 지식 표현 시스템인 Sphinx는 인공지능의 기본 도구로 사용할 수 있는데, 구체적으로 전문가 시스템 및 지능적 정보 검색 시스템을 개발하는데 사용할 수 있다. 병렬 추론 기계 X-WAM은 인공지능 개발을 위한 지능형 컴퓨터의 추론 기능을 담당하는 서버로서 사용될 수 있으며, 또한 기호처리를 위한 범용 병렬처리 시스템의 일부분으로도 사용될 수 있다.

본 연구를 통하여 자체 개발한 병렬 추론 기계이 상용화 되기까지는 계속적인 지원이 필요하며, 또한 성능 향상을 위하여 프로세서의 갯수를 수십개 이상으로 늘리기 위하여 추가적인 예산이 필요하다.

# SUMMARY

This project aims at the realization of the parallel computer system which can process a wide range of knowledge information. In knowledge information processing, most programs are large and complex. Logic language is chosen to be a basic language, since it provides logical reasoning and high degree of parallelism.

To achieve this goal, a parallel inference machine, called X-WAM, is designed and implemented, and a hybrid knowledge representation system, called Sphinx, is realized. X-WAM is a combined machine of XWAM-I which is an AND parallel machine and XWAM-II which is an OR parallel machine. A prototype X-WAM, consisting of 8 processors and host machine, is built and demonstrated. The result shows the increase of about 5 to 10 times in performance than the single processor machine. Sphinx is the system which combines the classification-based reasoning with the theorem prover using the logic programming technique. In this year, we have finished the implementation of Sphinx and made some knowledge base programming environments.

Another research result we have achieved is an extension of logic language. To increase the expressive power of logic language, we have designed a functional logic language called Aflog, and designed an abstract machine for it.

# 여 백

# 목 차

# Table of Contents

# 제 1 부

# 병렬 추론 컴퓨터에 관한 연구

# 여 백

# 제 1 장 서 론

병렬 추론 컴퓨터에 관한 연구는 1980년대 초에 일본의 ICOT에서 대량의 지식 정보를 병렬로 처리하는 제5세대 컴퓨터 프로젝트를 시작한 후, 유럽과 미국에서 이와 유사한 연구를 수년간 계속해와서, 이제는 그 시제품이나 상용화된 제품이 등장하고 있는 추세이다. 이들은 대부분 논리언어를 병렬로 수행시킬 수 있는 자신들 고유의 수행모델을 제안하고, 이를 상용 멀티프로세서 시스템이나 전용 멀티프로세서를 이용하여 구현하는 연구를 수행 중에 있다.

ICOT에서는 제 5세대 컴퓨터 프로젝트의 첫번째 단계 (1982년 ~ 1984년)에서 순차 추론 머신인 PSI를 개발하였으며 두번째 단계(1985년 ~ 1988년)에 PSI를 64개 연결한 병렬 추론 기계인 Multi-PSI를 개발하였다. 유럽에서는 일본의 제 5세대 컴퓨터 프로젝트가 시작된 비슷한 시기에 ESPRIT 프로젝트가 시작되었는데, 이 프로젝트를 중심으로 하여 영국의 Imperial College, 서독의 ECRC, 스웨덴의 SICS 연구소에서 병렬 추론 컴퓨터를 개발해 왔다. 특히 SICS에서는 상용 병렬 컴퓨터인 Encore Multimax에서 병렬 추론 시스템을 개발하고 있으며, 추론기능뿐만 아니라, 일반 병렬 컴퓨터로도 사용가능한 Data Diffusion Machine등을 개발하고 있다. 미국에서도 1980년대초에 컴퓨터와 반도체 분야의 신기술을 연구하는 연구소가 여러개 설립되었는데, 이중 MCC에서는 다량의 지식을 처리할 수 있는 CYC 프로젝트가 진행중이다.

본 연구는 이러한 세계적인 추세에 맞추어 논리 언어를 위한 병렬 추론 머신을 개발하기 위하여 시작되었다. 본 연구에서 개발한 병렬 추론머신 X-WAM(eXtended Warren Abstract Machine)은 논리언어의 두가지 대표적인 병렬성인 AND 및 OR 병렬성에 각각 바탕을 둔 XWAM-I과 XWAM-II로 나누어져 연구되어 왔다. XWAM-I은 AND 병렬성을 추구하고 있는 Goal Process 모델[Cho87]을 구현한 것이며, XWAM-II[Cho88]는 OR 병렬성을 추구하는 모델로서 일반적으로 OR 병렬성을 구현하는데 소비되는 과다한 오버헤드를 줄이고자 컴파일시에 프로그램의 분할이 이루어지며, 분할된 프로그램들을 여러 프로세서에서 동시에 수행한다.

본 연구에서는 1,2차년도의 연구를 바탕으로 하여 XWAM-I과 XWAM-II를 결합하여 AND 및 OR 병렬성을 동시에 지원할 수 있는 실용적인 모델 X-WAM을 제안하고 이의 성능을 분석하였으며, 또한 상용 마이크로 프로세서를 이용하여 XWAM-II모델에 기초한 프로토타입 병렬 추론 시스템을 구현하였다. 그리고 논리언어의 병렬 코드를 생성하기 위한 컴파일러, 어셈블러 및 에디터 등을 포함한 사용자 프로그래밍을 위한 환경을 개발하였으며, 실제 사용될 수 있는 시스템으로 만들기 위하여 Prolog에

데이타베이스(recorda, recordz, recorded 등)와 입출력(put, get, see 등)을 위한 기능등을 첨가하였다. 또한 X-WAM의 한 프로세서 모듈로 사용될 수 있는 전용 Prolog 프로세서 APP를 설계하였다. 그리고 이런 연구와는 별도로 논리 언어와 함수 언어의 특징을 결합하여, 한 프로그램 안에서 함수 언어 형태의 프로그래밍과 논리 언어 형태의 프로그래밍을 동시에 할 수 있는 함수 논리 언어에 대한 연구도 수행하였다. 본 보고서에서는 이러한 연구 결과들에 대하여 각각 설명하고자 한다.

본 보고서는 다음과 같이 구성되어 있다. 2장에서는 AND 및 OR 병렬성을 동시에 추구하는 X-WAM의 설계와 시뮬레이션 결과를 이용한 성능 평가에 대하여 설명하고, 3장에서는 실제 프로토타입으로 구현한 병렬 추론 머신 X-WAM의 설계 및 구현에 대하여 설명한다. 4장에서는 X-WAM의 기본 프로세서 모듈로 사용할 수 있는 전용 Prolog 프로세서 APP의 설계에 대하여 설명하고, 5장에서는 함수 논리 언어 Aflog를 위한 추상 머신인 F-WAM에 대하여 설명한다. 마지막으로 제 6장에서는 결론 및 앞으로의 연구 방향을 제시하였다.

# 제 2 장 X-WAM 다중 처리기 시스템

논리 언어에 내재한 AND 병렬성과 OR 병렬성은 대부분의 논리 언어를 위한 병렬 수행 모델에서 추구하는 중요한 병렬성이다[Con83, CiH84, GTM84, ISK84, Kim86, LiM86]. 하지만 이런 두개의 병렬성을 모두 추구하는데는 많은 오버헤드가 들기 때문에, 최근에 제안된 병렬 수행 모델들은 두 병렬성 중에서 하나만을 추구하는 경향이 많다.

AND 병렬성을 추구하는데 있어서의 문제점은 수행시에 여러 AND 프로세스사이에 있는 자료 종속관계를 제어하는데 드는 오버헤드이며, 최근에 제안된 AND 병렬 수행 모델들은 이런 오버헤드를 줄이는 방법을 제시하고 있다 [Bor84, DeG84, ChD85, DeM85, Her86a, LKL86, KLP88]. [Cho87]에서 제시한 고올 프로세스 모델(GPM : Goal Process Model)도 이런 AND 병렬 수행 모델 중의 하나이다.

OR 병렬성을 추구하기 위해서 드는 오버헤드는 새로운 OR 프로세스를 생성할 때 바인딩 환경(binding environment)을 복사하는데 드는 오버헤드이며, 이런 오버헤드를 줄이기 위하여 대부분의 OR 병렬 모델은 여러 프로세서가 메모리를 공유하며 같은 주소 영역을 갖는 공유 메모리 다중 처리기 시스템(shared memory multiprocessor system)을 가정하고 있다 [Ove85, War87, CiH84]. 하지만 이런 방법은 공유 환경을 관리하는 새로운 문제가 발생하게 된다 [Con87, LeK88].

한편 Clocksin은 논리 프로그램을 나타내는 AND/OR 탐색 트리를 분할하여 탐색하는 방법, 즉 하나의 연역 패스(deduction path)를 하나의 프로세서에게 할당하는 DelPhi 원리와, 이를 이용한 동적 OR 병렬 모델을 제안하였다. 이 모델에서 각 프로세서는 서로 다른 연역 패스를 탐색하기 때문에 서로 통신을 할 필요가 없게 된다 [Clo87]. 그러나 이 모델의 문제점은 모든 프로세서를 관리하는 모니터 프로세서에게 통신이 집중하게 되며, 탐색 트리를 동적으로 분리하는 것이 성능 저하의 주요 요인이 된다. 또한, 프로세서간의 통신이 없는 대신 같은 계산을 중복해서 해야 하며, 이로 인하여 컷(cut)이나 데이타베이스 처리용 술어, 혹은 입/출력 술어와 같이 부수 효과(side effect)가 있는 술어를 처리하기가 어렵다[LeK88]

본 연구의 선행연구[Cho88]에서 Clocksin의 OR 병렬 모델의 문제점을 해결하기 위하여, Warren이 설계한 순차 추론 머신(WAM)에 기초를 둔 정적 OR 병렬 처리 모델 XWAM-II를 제안하였다. 이 모델에서는 논리 프로그램을 컴파일시에 서로 다른 독립적인 부분으로 나누고, 나누어진 각 부분을 하위의 한 $WAM^+$ 프로세서에서 수행한다.

WAM$^+$은 정적 OR 병렬 처리 모델에 필요한 인스트럭션을 첨가한 것으로서, WAM의 확장형이다.

본 연구에서는 이와 같은 선행 연구의 결과를 이용하여 Goal Process Model(XWAM-I)의 AND 병렬성과 XWAM-II에서의 OR 병렬성을 동시에 추구하는 병렬 추론 머신 **X-WAM**(eXtended Warren Abstract Machine)을 제안한다.

본 장은 다음과 같이 구성되었다. 2.1절에서 X-WAM 시스템의 선행 시스템인 XWAM-I과 XWAM-II를 개략적으로 소개한다. 그리고, 2.2절에서 X-WAM 시스템의 개념적인 구조에 대하여 알아보고, 2.3절에서는 주어진 프로그램을 정적으로 분할하는 방법에 대하여 설명하겠다. 2.4절에서는 Prolog 프로그램을 X-WAM에서 수행시키는 방법에 대하여 알아보고, 2.5절에서는 시뮬레이션에 의한 성능 평가를 기술하였다. 마지막으로 2.6절에서 본 장을 요약한다.

## 2.1 배 경

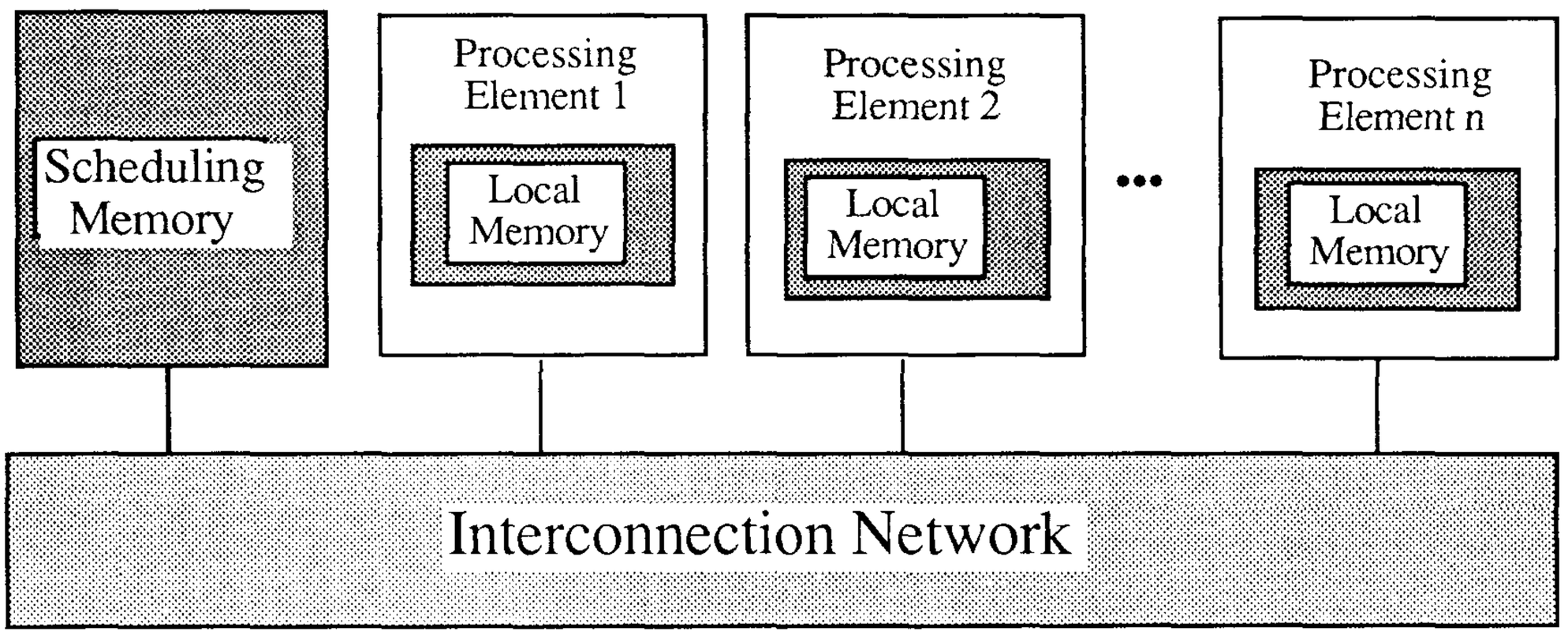
본 장에서는 본 연구의 선행 연구인 XWAM-I과 XWAM-II를 개략적으로 설명한다.

### 2.1.1 XWAM-I 시스템: 고울 프로세스 모델[Cho87]

고울 프로세스 모델은 논리 언어의 AND 병렬성을 추구하는 모델이며, 하위 컴퓨터 구조로서 별개의 지역 메모리를 가지는 프로세서가 임의의 상호 연결망(interconnection network)으로 서로 연결되어 있는 구조를 가정한다. 그리고 각 프로세서들은 메시지를 이용하여 서로 통신하도록 하며, 프로세서와 고울 프로세스 사이에는 1 : 1 대응관계를 갖도록 하고 있다 [Kim88b].

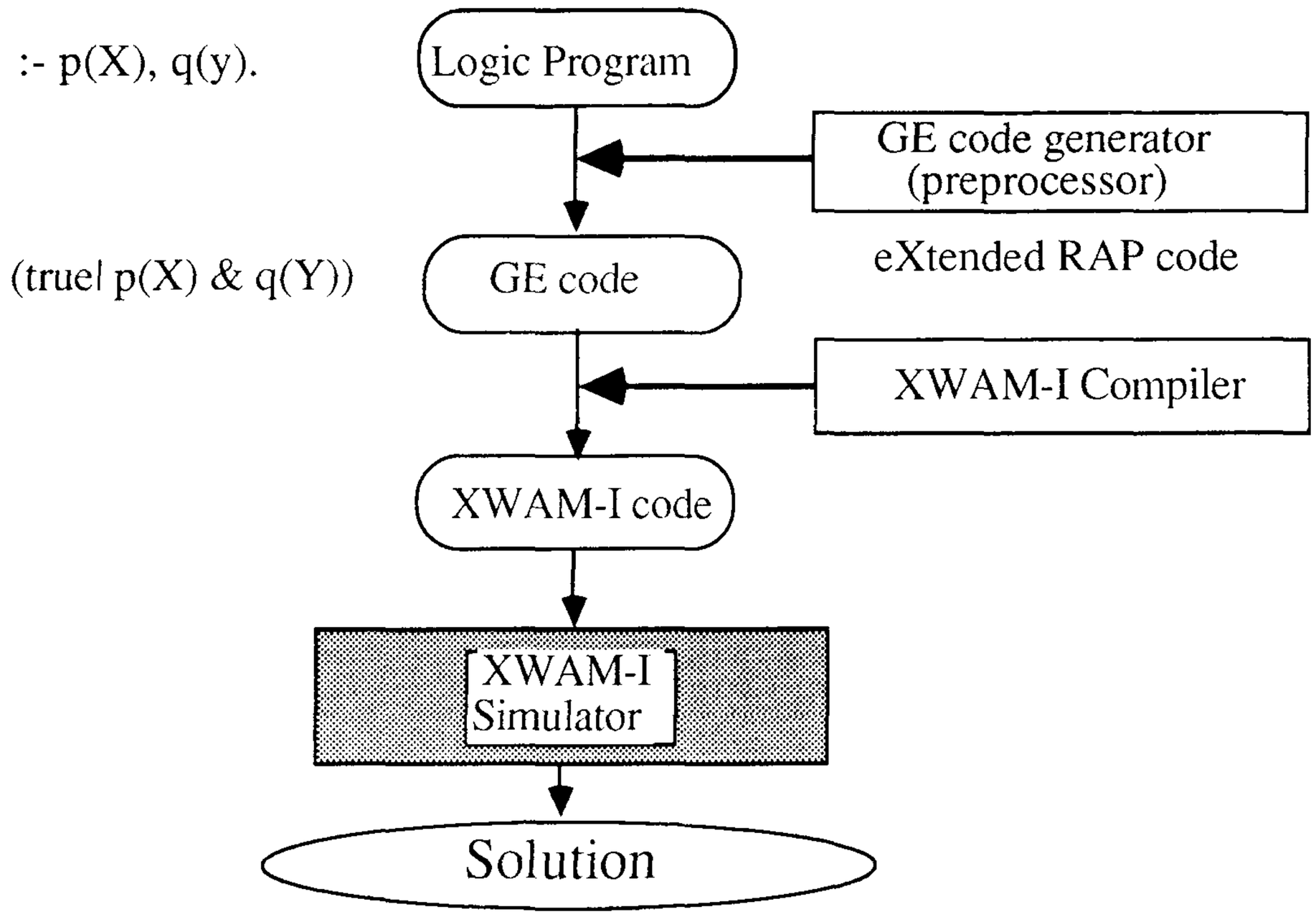
XWAM-I은 위와 같은 구조를 기초로 하는 추상적인 다중 처리기 시스템이다. XWAM-I의 각 프로세서는 WAM을 바탕으로 하고 있으며, 전체 시스템은 제한된 AND 병렬 수행 방식인 동시에, eager evaluation과 요구에 의한 파이프 라인식 OR 병렬 수행 방식으로 수행한다. <그림 2.1>은 XWAM-I의 하드웨어 개관을 나타낸 것으로, 그림에서 스케쥴링 메모리(SM)는 모든 프로세서가 접근 가능한 메모리로서 AND 병렬성을 지원하기 위한 목적으로 사용되며, 각 프로세서의 부하 상태에 대한 간단한 정보만을 갖고 있다. AND 병렬 처리 서브 고울을 생성한 프로세서는 자신의 부하, 즉 AND 병렬 처리 서브고울의 갯수를 스케쥴링 메모리에 기록한다. 쉬고 있는 프로세서는 스케쥴링 메모리로부터 다른 프로세서의 부하에 대한 정보를 읽어낸 다음 부하가 가장 높은

프로세서에게 메시지를 전달하여 그 프로세서로부터 AND 병렬 서브 고울을 이전받아 처리하게 된다. 이와 같은 프로세스의 할당 정책은 간단하여 거의 오버헤드를 요구하지 않으며 또한 쉬고 있는 프로세서가 가장 부하가 높은 프로세서로부터 고울을 이전받음으로써 가능한 프로세서간의 부하를 균형상태로 유지하려고 한다.




<그림 2.1> XWAM-I의 하드웨어 개관

XWAM-I은 WAM을 기초로 한 다른 병렬 처리 Prolog 머신 [Her86a]처럼 논리 프로그램의 컴파일 기법을 적용하였다. 주어진 Prolog 프로그램을 분석하여 AND 병렬 수행 가능한 서브 고울들을 찾아내는 전 처리단계를 거친 후, 컴파일된 코드를 XWAM-I이 실행하게 된다. <그림 2.2>는 XWAM-I의 이러한 수행 단계를 나타낸 것이다. 전 처리 단계에서 생성되는 인터페이스인 그래프 표현식(graph expression : GE)은 [Kim86]에 자세히 설명되어 있으며, 컴파일 코드인 XWAM-I의 추상 인스트럭션 집합은 [Cho87]에 자세히 설명되어 있기 때문에 생략하도록 한다.




<그림 2.2> XWAM-I에서 프로그램 수행 단계

### 2.1.2 XWAM-II 시스템[Cho88]

XWAM-II 시스템은 논리 언어의 OR 병렬성을 추구하며, 다른 OR 병렬 모델과의 근본적인 차이는 정적 분할 기법을 사용한다는데 있다. 이 시스템 역시 WAM에 기반을 둔 다중 처리기 시스템이며, 각 프로세서는 논리 프로그램의 독립적인 단위인 OR 번치(bunch)를 탐색하기 위한 기능을 기존의 WAM에 첨가하여 확장한 것으로 $WAM^{+}$로 불린다. XWAM-II 시스템은 <그림 2.3>에 나타낸 것과 같이 모니터(Monitor), 분배망(Distribution Network), 그리고 추론 모듈(Inference Module)의 세 부분으로 이루어져 있다.

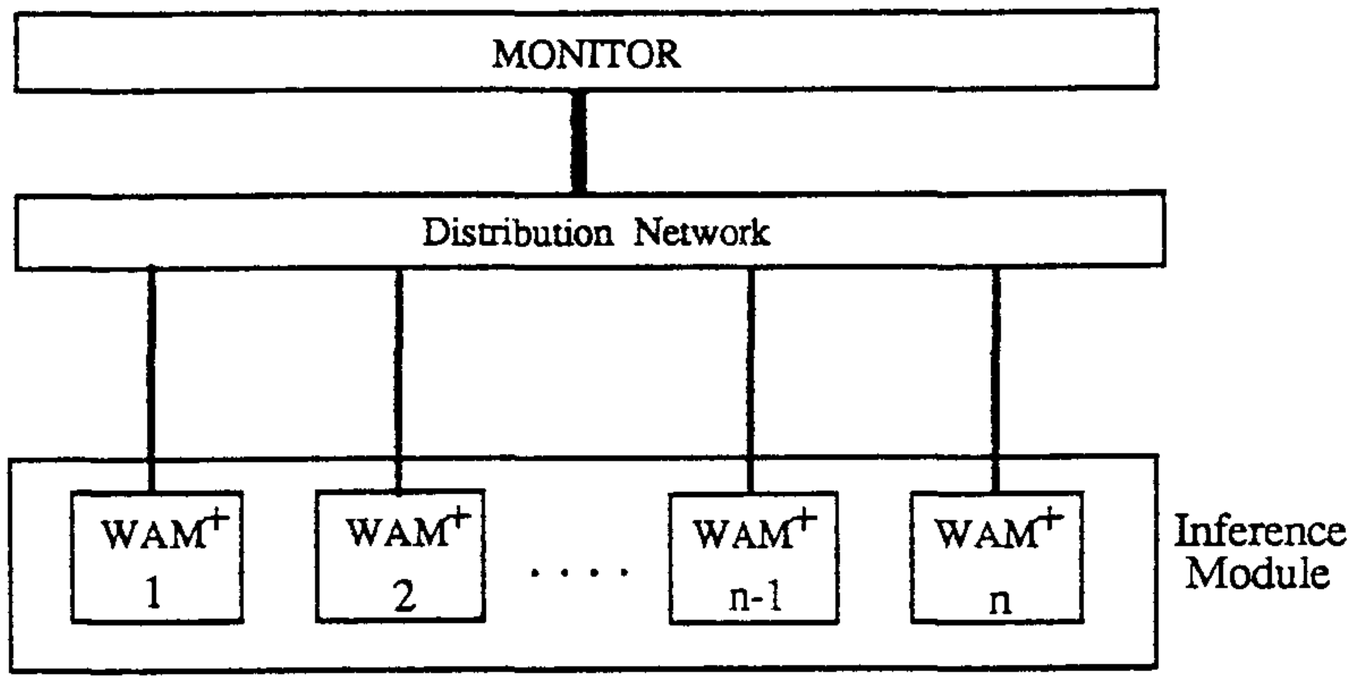
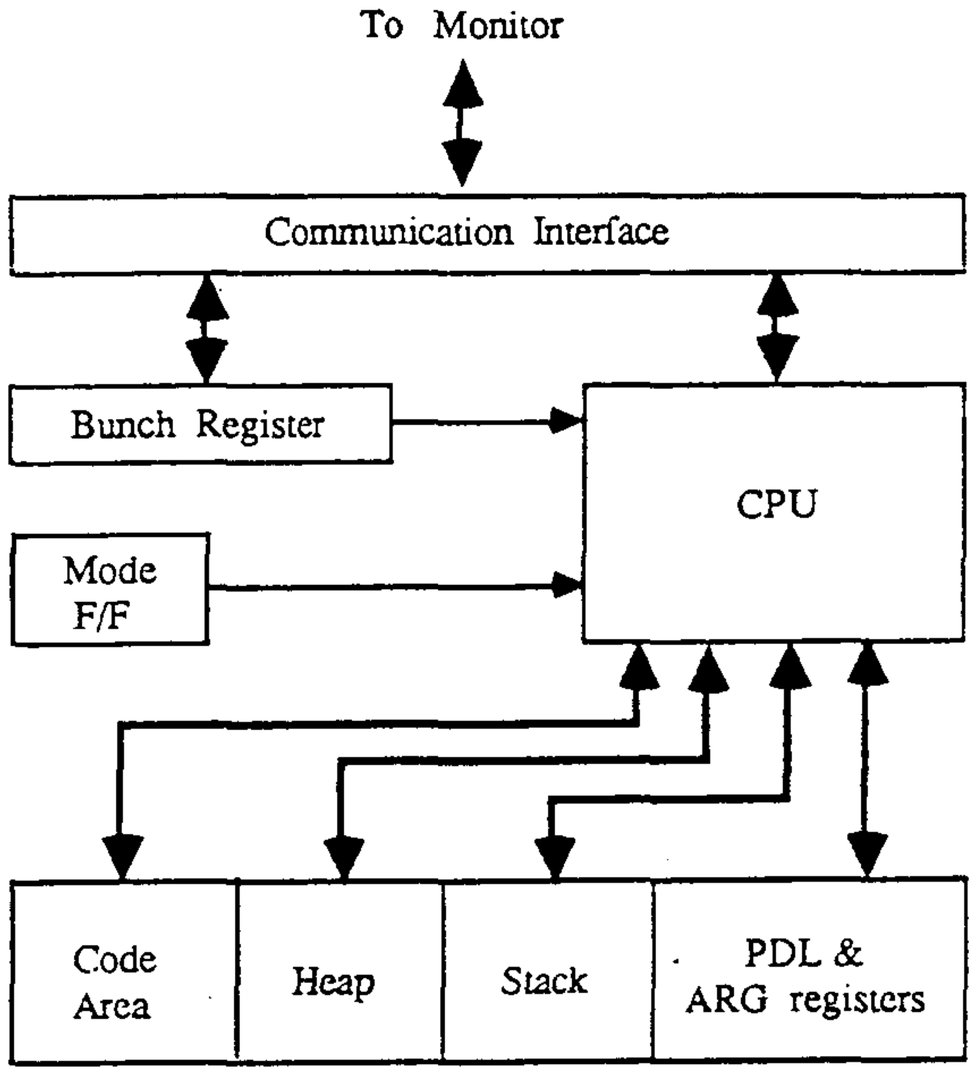



그림 2.3 XWAM-II 시스템 구조

모니터는 전체 시스템을 제어하고 사용자 인터페이스를 제공하는 전위 모듈(front-end module)이다. 즉, 입력 논리 프로그램을 정적으로 분석하여 $WAM^+$ 기계어 코드를 생성하고, 이 코드를 분배망을 통해 추론 모듈의 각 $WAM^+$ 프로세서들에게 분배하여 수행 시킨 후, 해들을 수집하여 사용자에게 보고한다. 또한 모니터는 사용자에게 편집기를 비롯한 여러가지 프로그래밍 환경을 제공한다. 분배망은 모니터와 추론 모듈사이의 통신을 제공하는 연결 네트워크로서 동적인 통신이 없기 때문에 간단한 위상 구조를 갖는 네트워크, 즉 버스, 성형 네트워크(star network), 링 네트워크(ring network) 등으로 구현할 수 있다. 추론 모듈은 논리 프로그램을 실제로 수행하는 모듈로서 많은 $WAM^+$ 프로세서들로 구성되어 있다.

각 $WAM^+$ 프로세서는 독립적으로 OR 번치들을 탐색하므로 서로간의 통신은 필요하지 않고 단지 모니터와 통신을 하면 된다. $WAM^+$의 내부 구조는 <그림 2.4>에 나타낸 것과 같이 기존의 WAM 구조에 OR 번치를 독립적으로 탐색하기 위하여 번치 레지스터(bunch register), 비트 검사기, 그리고 수행 모드를 나타내는 모드 플립플롭 등이 추가 되었다. 또한 모니터와의 통신을 위한 통신 인터페이스가 있다. $WAM^+$에 추가된 새로운 인스트럭션은 번치 레지스터의 비트에 따라 좌, 혹은 우로 브랜치하는 entry_point

인스트럭션 뿐이다. 이 인스트럭션의 의미는 [Cho88]에 자세히 기술되어 있다.

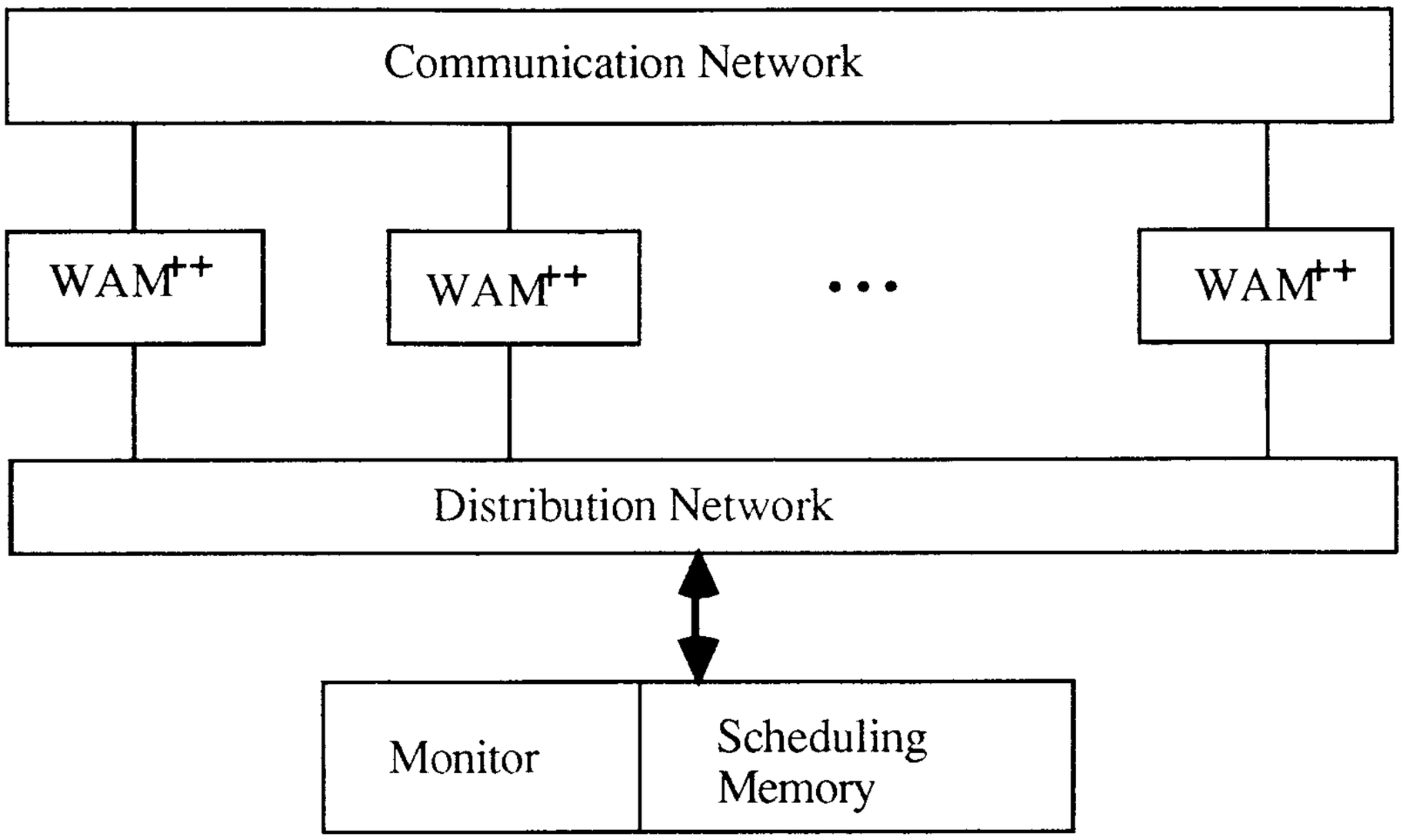



<그림 2.4> WAM$^{+}$ 구조

## 2.2 X-WAM의 시스템 구조

본 절에서는 앞절에서 설명한 AND 병렬성을 추구하는 XWAM-I과 OR 병렬성을 추구하는 XWAM-II를 결합한 새로운 수행 모델 X-WAM에 대하여 설명하도록 한다. X-WAM은 기본적으로 먼저 논리 언어 프로그램의 OR 병렬성을 이용하여 수행하다가, 쉬고 있는 프로세서가 생기면 AND 병렬성을 이용하여 수행하는 모델이다.

<그림 2.5>에 나타낸 것과 같이 X-WAM의 시스템 구조는 모니터와 분산 네트워크, 통신 네트워크, 스케쥴링 메모리, 그리고 WAM$^{++}$ 프로세서들로 이루어져 있다. 모니터는 입력 프로그램을 정적으로 분할하는 일과, 이것을 이용하여 X-WAM 코드를 생성하는 일을 담당하며, 이 생성된 코드는 분산/통신 네트워크를 이용하여 각 프로세서에게 나누어진다. WAM$^{++}$ 프로세서는 XWAM-I의 프로세서 기능과 XWAM-II의 WAM$^{+}$ 기능을 합친 프로세서로서 X-WAM 코드를 수행한다.




<그림 2.5> 제안된 X-WAM의 구조

코드의 분산이 이루어진 후에는 각 WAM++가 독립적으로 X-WAM 코드를 수행할 수 있기 때문에 정적 OR 모델을 제공하기 위하여 추가로 필요한 제어는 없다. 하지만 GPM에 기초를 둔 AND 병렬 수행의 가능성이 남아 있다. 이것은 수행 도중에 쉬고 있는 WAM++가 생기게 되면, 이 프로세서는 부하가 많은 WAM++로부터 AND 병렬 고울을 가지고 와서 수행하게 된다. 그러므로 AND 병렬 고울을 가지고 있는 프로세서는 주어진 코드를 다른 프로세서와 함께 수행할 수 있다. 최악의 경우 AND 병렬 고울을 가지고 있어도 쉬는 프로세서가 없으면 혼자 수행해야 하는 경우가 생길 수도 있다. 스케쥴링 메모리는 X-WAM에 있는 각 프로세서에 대한 부하 정보를 가지고 있어서, 위에서 언급한 AND 병렬 수행을 할 수 있게 한다.

## 2.3 Prolog 프로그램에 대한 정적 분할 기법

Prolog 프로그램은 논리 언어에 있는 비결정성을 이용하여 여러개의 서로 다른 프로그램으로 나누어질 수 있다. 다음 예제를 이용하여 설명하도록 한다.

예제 2.1:

```
p(X,Y):- q(X), r(X,Y).
q(a).    q(b).
r(a,b).  r(b,d).
```

예제 2.1에 있는 Prolog 프로그램은 대안 논리절을 나누어 4개의 서로 다른 프로그램으로 분할될 수 있다. 분할된 프로그램을 헤드 리터럴의 집합으로 나타내면 다음과 같다. 즉, {p(X,Y), q(a), r(a,b)}, {p(X,Y), q(b), r(a,b)}, {p(X,Y), q(a), r(b,d)}, {p(X,Y), q(b), r(b,d)} 등의 4개의 프로그램이 된다. 만약 2개의 대안 논리절을 가지고 있다면 2개의 서로 다른 프로그램으로 분할하여 독립적으로 수행할 수 있다.

이와 같은 Prolog 프로그램의 정적 분할은 여러가지 장점을 가지고 있다. 첫째로, 대부분의 OR 병렬 수행 모델에서 발생하는 수행시 오버헤드 없이 OR 병렬 수행을 할 수 있으며, 또한 분할된 프로그램에서 백트랙킹의 수가 적게 된다. 마지막으로 이런 방법은 비교적 간단한 방법이기 때문에 다중 처리기 시스템을 쉽게 만들 수 있다.

하지만 정적 OR 분할 모델은 다음과 같은 단점도 가지고 있다. 즉, 아주 큰 프로그램을 분할하면, 많은 수의 분할 프로그램이 생성되게 된다. 또한 되돌림(recursion)을 이용하는 프로그램은 입력의 크기 없이는 컴파일시 분할 할 수 없으며, 분할한다는 것 자체도 의미가 없는 일이다. 마지막으로 분할 된 프로그램이 현재 사용 가능한 프로세서의 갯수로 제한된다면, 프로그램을 어떻게 분할할 것인가 하는 문제도 발생하게 된다.

본 연구에서는 이런 문제를 경험적 정보(heuristics)를 이용하여 해결하였다[Cho88]. 이 방법에서의 기본적인 가정은 분할된 프로그램의 갯수는 현재 시스템에서 이용 가능한 프로세서의 갯수로 제한한다는 것이다. 분할된 프로그램은 번치(bunch)라고 부르는데, 이 하나의 번치는 여러개의 연역 패스를 포함하게 된다. 예제 2.1을 다시 생각하여 보자. 4개의 분할된 프로그램은 ${}_4C_2$ 방법을 이용하여 2개의 번치로 묶을 수 있다. 이런 번치 중에서 2개의 번치를 경험적 지식을 이용하여 선택할 수 있다. 이렇게 여러개의 번치 중에서 적당한 번치를 선택하는 경험적 지식은 다음과 같다.

o 프로그램의 연역 트리는 밑부분과 왼쪽을 먼저 기준으로 하여 분할한다. 이것은 밑 부분과 왼쪽 부분의 백트랙킹 길이를 짧게하기 위함이다.

o 리터럴의 대안(alternative) 논리절은 그들의 인수 분석의 결과에 의하여 순서가 바뀌어지게(shuffling) 된다. 이런 순서 바꿈은 그들 인수의 입출력 모드에 기초를 두어 수행한다. 이런 경험적 정보는 대안 논리절을 해싱하는 효과와 각 OR 번치를 같은 크기로 만들 수 있게 한다.

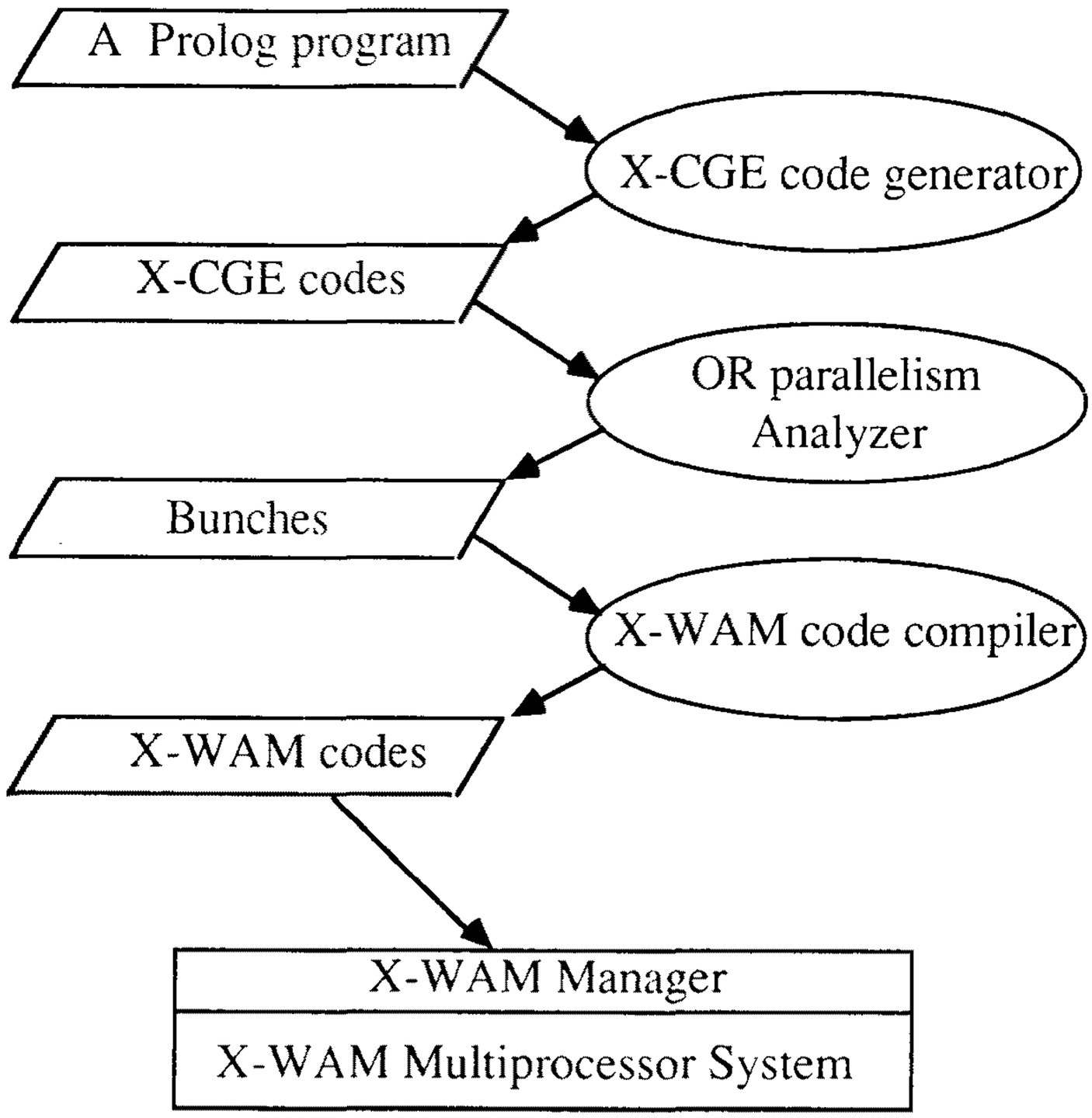
o 되돌림을 이용하여 정의된 논리절은 대게 결정적인 특징을 가지고 있고, 더구나 이런 부분을 분할하는 것은 어렵기 때문에 분할하지 않는다.

이런 경험적 정보에 대한 자세한 내용은 [Cho88]에 자세히 설명되어 있다. 이런 경험적 정보를 이용하면 예제 2.1에 있는 프로그램은 다음과 같은 2개의 번치로 나눌 수 있다. 즉, {p(X,Y), q(a), r(a,b), r(b,d)}와 {p(X,Y), q(b), r(a,b), r(b,d)}으로 나눌 수 있다.

위에서 언급한 번치를 풀기 위해서는 두개의 수행 모드가 필요하다. 하나는 "패스 모드"이며, 또 다른 하나는 "트리 모드"이다. 패스 모드인 경우에 번치는 번치의 AND-OR 탐색 트리의 결정적인 패스를 탐색하게 되고, 트리 모드인 경우에는 깊이 우선의 방법으로 비결정적인 패스를 탐색하게 된다.

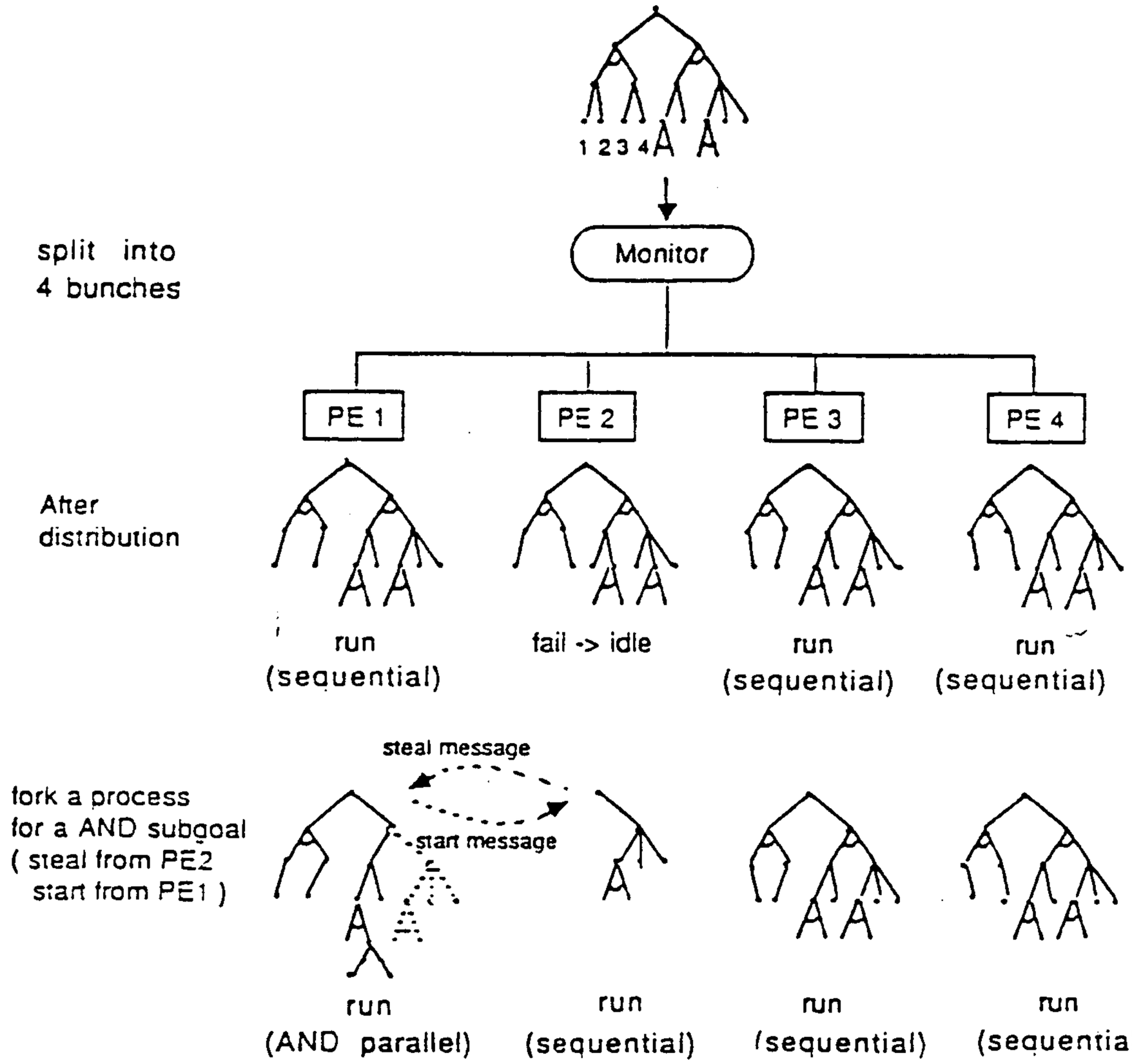
## 2.4 X-WAM에서의 Prolog 프로그램의 수행

Prolog 프로그램은 여러 단계를 거쳐서 X-WAM 코드로 컴파일된다. 시스템 소프트웨어의 전체적인 구성은 <그림 2.6>과 같다. 첫번째 단계는 주어진 프로그램을 분석하여 X-CGE 코드를 얻어내는 것이며, 두번째 단계는 이것을 여러개의 번치로 나누는 것이다. 이때 앞에서도 언급한 것과 같이 총 번치의 갯수는 현재 시스템에서 사용가능한 프로세서의 갯수로 제한한다. 마지막으로 각 번치는 X-WAM 코드로 컴파일된다.




<그림 2.6> X-WAM에서의 프로그램 수행 단계

X-WAM의 모니터는 컴파일된 X-WAM 코드를 시스템에 있는 모든 WAM$^{++}$ 프로세서에게 보내고, 각 WAM$^{++}$ 프로세서는 같은 코드를 수행하게 된다. GPM에서는 요구 동기(demand driven) 스케줄링 방법을 사용하고 있기 때문에, 한 번치의 AND 병렬 수행은 쉬고 있는 프로세서가 있을 때 시작된다. 각 프로세서에 대한 부하 정보는 스케줄링 메모리에 기억되고 계속적으로 수정된다. 만약 어떤 프로세서가 자신에게 할당된 코드의 수행을 끝내면, 그 프로세서는 스케줄링 메모리를 검사하여 수행할 AND 병렬 고울이 있는가를 살펴본다. 만약 AND 병렬 고울을 가지고 있는 프로세서가 있으면 쉬고 있는 프로세서는 AND 병렬 고울을 가지고 있는 프로세서에게 "steal" 메시지를 보내게 된다. 어떤 프로세서가 steal 메시지를 받으면 그 프로세서는 메시지를 보낸 프로세서에게 AND 병렬 수행 고울의 코드를 보내어 수행하도록 한다. <그림 2.7>은 X-WAM에서의 AND/OR 병렬 수행의 예제를 나타낸 것이다.




<그림 2.7> 정적 OR 병렬 수행과 동적 AND 병렬 수행

## 2.5 성능 평가

X-WAM에 대한 시뮬레이터는 4.2 BSD를 수행하는 SUN3/160에서 C 언어로 작성하였다. 각 인스트럭션에 대한 사이클 시간은 각 인스트럭션을 수행하는데 필요한 레지스터 참조 횟수와 메모리 참조 횟수의 합으로 측정하였다. 이밖에 리터럴의 인수 갯수(A), 디레퍼런스하는데 걸리는 시간(d), 단일화(unification)하는데 걸리는 시간(u) Par_Frame에 있는 병렬 서브고울의 갯수, 그리고 리터럴의 갯수(L)도 수행시간을

측정하는데 사용되었다.

메모리를 참조하는데 걸리는 시간(M)은 레지스터를 참조하는데 걸리는 시간(R)의 비율로 측정하였는데, 본 시뮬레이션에서는 메모리를 참조하는 시간이 레지스터를 참조하는 시간보다 3배 더 걸린다고 가정하였다. 즉, M = 3*R이라고 가정하였다. 디레퍼런스를 하는 시간과 단일화를 하는 시간은 현재의 상태에 따라서 동적으로 변하기 때문에, d와 u는 수행시에 측정하였다. 프로세서들 사이에서의 메시지를 전달하는데 드는 오버헤드를 측정하기 위하여, 크기가 S인 메시지를 전송하는데 드는 시간을 3*M*S라 가정하였다. 각 인스트럭션에 대한 예상 수행 시간은 <부록-1>에 첨가하였다. X-WAM에 대한 성능 평가는 AND 병렬 수행을 하는 경우와, OR 병렬 수행을 하는 경우, 그리고 AND와 OR 병렬 수행을 모두하는 경우의 3가지로 측정하였다.

### 2.5.1 AND 병렬 수행에 대한 성능 평가

AND 병렬 수행에 대한 성능을 향상하는 방법은 서로 독립적인 AND 서브 고울을 동시에 수행하는 RAP(Restricted AND Parallelism)을 이용하는 방법과, OR 대안절을 먼저 수행하는 EO(Eager OR) 방법, 그리고 인텔리전트 백트랙킹 방법(IB : Intelligent Backtracking) 등 3가지가 있다. 다음과 같은 4가지 경우에 대하여 성능 평가를 하기로 한다.

i) RAP을 이용하면서 기초적인(naive) 백트랙킹을 사용하는 경우(RAP),
ii) RAP과 인텔리전트 백트랙킹을 사용하는 경우 (RAP + IB),
iii) RAP과 대안절을 먼저 수행하는 경우 (RAP + EO),
iv) RAP과 EO를 인텔리전트 백트랙킹과 함께 사용하는 경우 (RAP + EO + IB)

총 5개의 프로그램을 벤치마크 프로그램으로 선택하였는데, 이 프로그램들은 서로 다른 특성을 가지고 있는 프로그램들이다. 벤치마크에 사용된 프로그램들은 <부록-2>에 첨가하였다. arch 프로그램은 [Kow79]에서 발췌하였고, map coloring 프로그램은 [Con83]에서 발췌하였다. [Her86a]에서 발췌하여 수정한 check 프로그램은 RAP만 사용했을 때와 EO를 함께 사용하였을 때의 성능을 평가하기 위한 것이다. arch 프로그램과 map coloring 프로그램은 논리절들 사이의 인텔리전트 백트랙킹을 할 수 있는 프로그램이며, fibonacci 프로그램은 인텔리전트 백트랙킹이 필요없는 프로그램이지만 많은 AND 병렬성을 가지고 있는 프로그램이다.

또한 RAP만 사용하는 경우와 비교하여 얼마만큼의 오버헤드가 더 필요하는지를 측정하였다. 인텔리전트 백트랙킹을 위한 오버헤드로는 BE(Backtrack Environment)를 만드는데 사용되는 오버헤드와 단일화(unification) 동작을 수행하는데 드는 오버헤드, BE를 액세스 하는데 드는 오버헤드, 그리고 실패의 이유를 분석하는데 드는 오버헤드 등의 합이다. EO를 하는데 드는 오버헤드로는 어떤 고울에 대한 환경을 다른 프로세서로 복사하는데 드는 오버헤드 뿐이다.

check 프로그램은 두 가지 방법으로 성능을 평가하였는데, 10번의 되돌림(recursion)을 하는 times10과 20번의 되돌림을 하는 times20이 그 두가지이다. 이 times10과 times20은 되돌림 하는 횟수만 틀린 프로그램인데, 이것을 이용하여 EO의 효과를 측정할 수 있다. <그림 2.8>과 <그림 2.9>에 나타낸 것과 같이 RAP만을 사용하였을 때의 성능 향상은 각각 1.66과 1.58인데, 이것은 다른 바인딩을 지연 계산한 결과이다. 즉, 한 프로세스가 redo 메시지가 오기 전까지 새로운 바인딩을 만들지 않기 때문이다.

하지만, 만약 프로세스가 redo 메시지가 오기 전에 새로운 바인딩을 먼저 만들면 성능은 향상될 수 있다. 이런 경우(RAP + EO)의 성능 향상은 times10인 경우에는 2.91이고, times20인 경우에는 3.54이다. <부록-3>에 있는 메시지 교환의 패턴은 RAP만을 사용한 경우와 RAP과 EO를 사용한 경우의 차이를 잘 나타내고 있다. 시뮬레이션을 통하여 얻은 또 다른 결과는 병렬 고울의 깊이(논리절의 호출)가 성능 향상을 결정짓는 중요한 요인이라는 것이다. 이와같은 두 프로그램의 경우 인텔리전트 백트랙킹은 성능 향상에 거의 도움이 되지 못한다.

arch 프로그램은 RAP과 EO와 비교하여 IB를 사용하였을 때 얻을 수 있는 효과를 잘 나타내는 프로그램이다. <그림 2.10>에 나타낸 것과 같이 RAP과 RAP+EO의 성능 향상은 백트랙킹이 많아질수록 감소함을 알 수 있었으며, 실패의 횟수는 <부록-4>에 나타내었다. RAP을 사용할 때의 최대 성능 향상은 4개의 프로세서를 사용할 때 1.1이며, 이런 성능은 백트랙킹을 위한 바인딩을 미리 수행함으로서 쉽게 향상될 수 있다. 하지만 이런 결과는 많은 백트랙킹이 바인딩을 미리 수행하는 것의 효과를 감소시키기 때문에 성능 향상에는 별로 도움이 안된다. <그림 2.10>에 나타낸 것과 같이 4개의 프로세서를 사용한 경우의 RAP+EO의 성능 향상은 1.3 정도이다. 하지만 IB를 사용하면 성능을 많이 향상시킬 수 있다. 즉, RAP+IB인 경우의 성능 향상은 5.14 정도이며 RAP+IB+EO인 경우에는 6.84 정도이다. 이와 같이 RAP+IB+EO인 경우에 많은 성능 향상이 되는 중요한 이유는 실패의 횟수를 줄임으로서 EO의 효과를 극대화할 수 있기 때문이다.

map coloring의 시뮬레이션 결과로 알 수 있는 사항은 RAP+IB의 경우가 다른 3가지 방법에 비교하여 최대의 성능을 낼 수 있다는 것이다. 이와같은 결과는 EO를 하는데 드는 오버헤드에서 기인한다. 즉, color 논리절 내의 서브 고울은 깊이가 얕기 때문에 EO의 효과가 거의 없게되어, 성능에 영향을 미치지 않는다. <그림 2.11>에 나타난 것과 같이 이 예제 프로그램의 RAP+IB의 경우 최대 성능 향상은 2.52이다. Map coloring은 arch 프로그램보다 더 많은 AND 병렬성을 가지고 있지만, 서브고울의 깊이가 얕고 백트랙킹의 평균 거리가 짧기 때문에 최대 성능은 arch 프로그램의 경우보다 나쁘다.

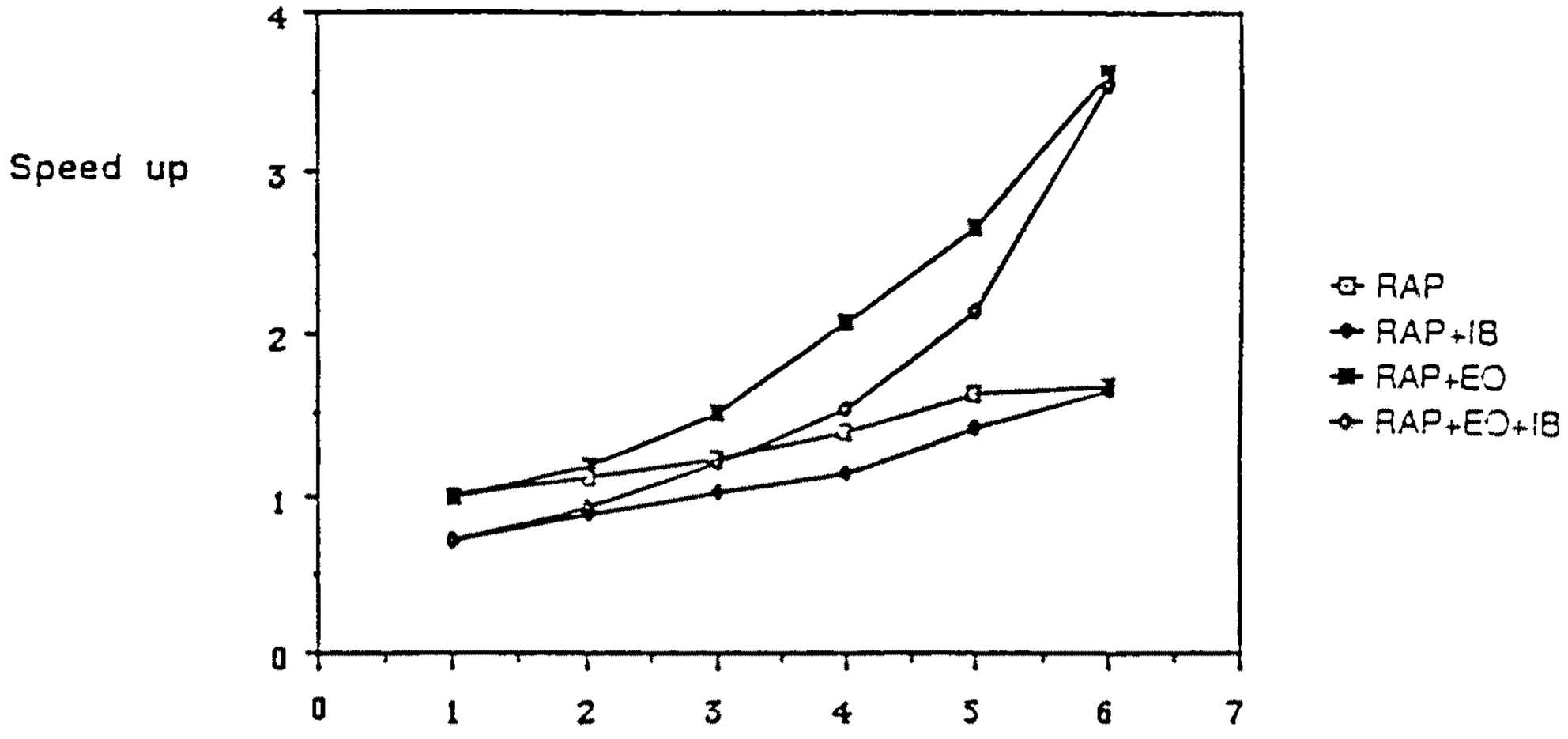
fibonacci 프로그램은 결정적인 특성을 가지고 있는 프로그램이다. 이와 같은 특성을 가지고 있는 프로그램의 경우에는 인텔리전트 백트랙킹이나 EO와 같은 방법이 거의 효과를 보지 못한다. 이런 프로그램을 벤치마크로 시뮬레이션함으로서 인텔리전트 백트랙킹과 EO를 하는데 오버헤드를 알아볼 수 있다. fibo(10,R)을 수행하는 경우에 IB를 하기 위하여 드는 오버헤드는 7.6% 정도이다. 하지만 이런 오버헤드는 프로세서의 갯수가 증가함에 따라 감소하게 된다. 그 이유는 이런 오버헤드가 여러 프로세서들 사이로 분산되기 때문이다. <그림 2.12>에 나타낸 것과 같이 RAP+IB+EO의 성능은 RAP만을 사용한 경우와 거의 비슷하게 된다. IB에 대한 오버헤드와 비교하여 EO에 대한 오버헤드는 매우 작게되는데, 그 이유는 fibo(N,R)의 인수를 복사하는데 디레퍼런스를 하지않아도 되기 때문이다. <그림 2.12>에서 할 일을 다 끝낸 프로세서는 제거(kill) 메시지가 도착하기 전까지 지연되기 때문에, 성능 향상 그래프가 계단 모양이 된다. 만약 프로세서가 제거 메시지 없이 병렬로 다른 고울을 풀 수 있다면, 그래프의 모양은 선형 증가형이 될 것이다.

### 2.5.2 OR 병렬 수행의 성능

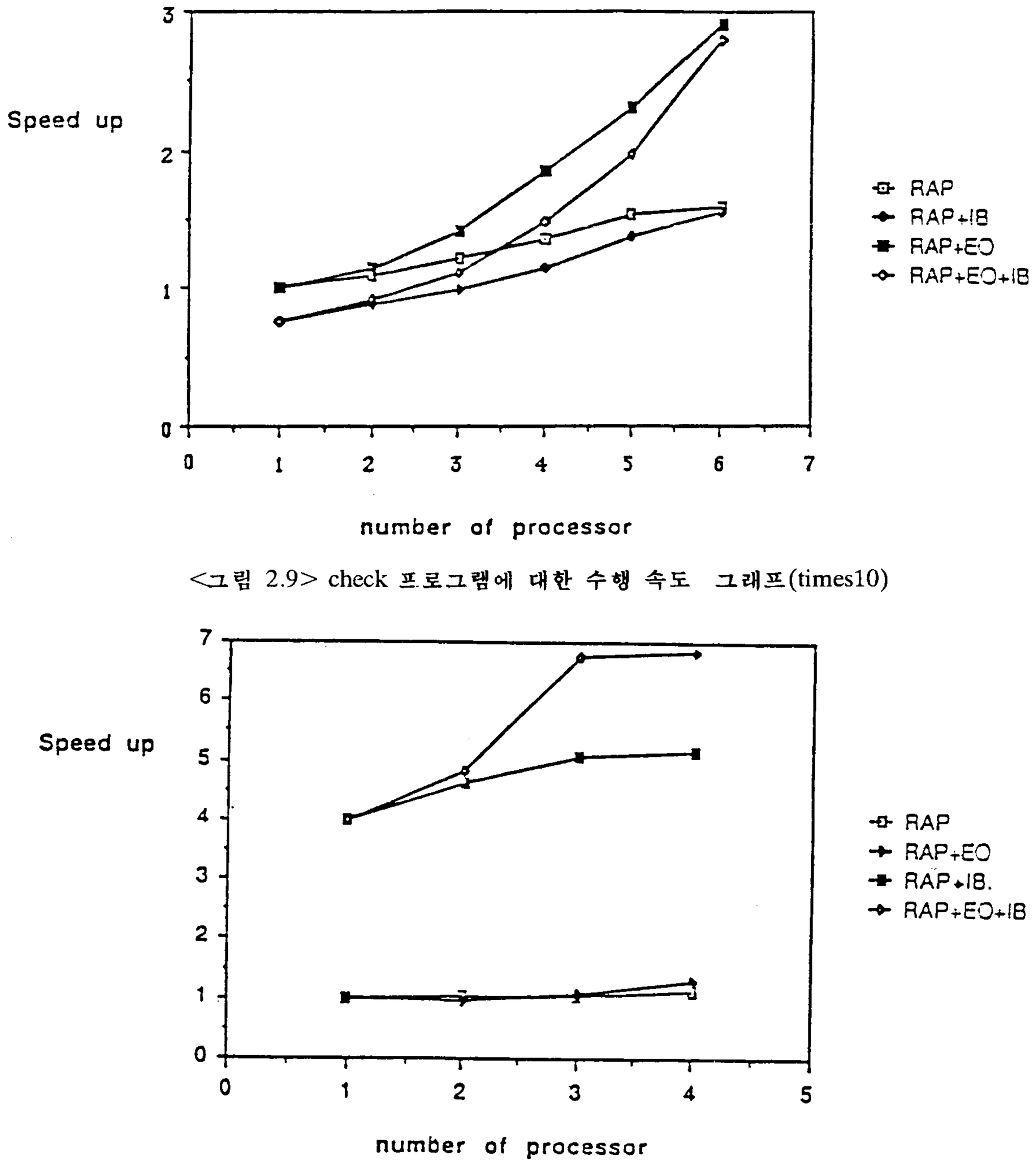
2.3절에서 언급한 것과 같이 주어진 프로그램은 OR 병렬 수행을 하기위하여 컴파일시에 번치라고 하는 단위로 분할된다. 각 프로세서는 독립적으로 이런 번치를 수행하고, 그 결과를 '성공'이나 '실패'메시지를 통하여 모니터에게 보고한다. 이런 OR 병렬 수행 경우의 성능을 <부록-2>에 첨가한 벤치마크 프로그램을 이용하여 측정하였다. <그림 2.13>은 각각 프로그램에 대한 성능을 나타낸 것이다.

얻어진 성능향상은 1배~17배 정도인데, qucik sort의 경우 결정적인 특성을 갖는 프로그램이기 때문에 가장 나쁜 성능이 나왔다. 결정적인 특성을 갖는 프로그램의 경우에는 성능을 향상시키기 위한 OR 병렬 수행 방법이 존재하지 않는다는 것은 널리 알려진 사실이다. 가장 좋은 성능을 나타내는 경우는 많은 백트랙킹을 포함하고 있는 프로그램이다. 즉, 비결정적인 특성을 가지고 있는 프로그램의 경우 OR 병렬 수행을 하면

좋은 성능 향상을 얻을 수 있다. 이와같은 프로그램으로는 map coloring 프로그램, 데이타베이스 질의 프로그램, 그리고 eight queen 프로그램 등이며, 이런 경우 프로그램의 비결정적인 부분이 16개의 번치로 나누어지고 백트랙킹의 횟수도 역시 분할 되기 때문에 OR 병렬로 수행할 경우 큰 성능 향상을 얻을 수 있다. 이에 대한 자세한 내용은 [LeK88]에 자세히 언급되어 있다.

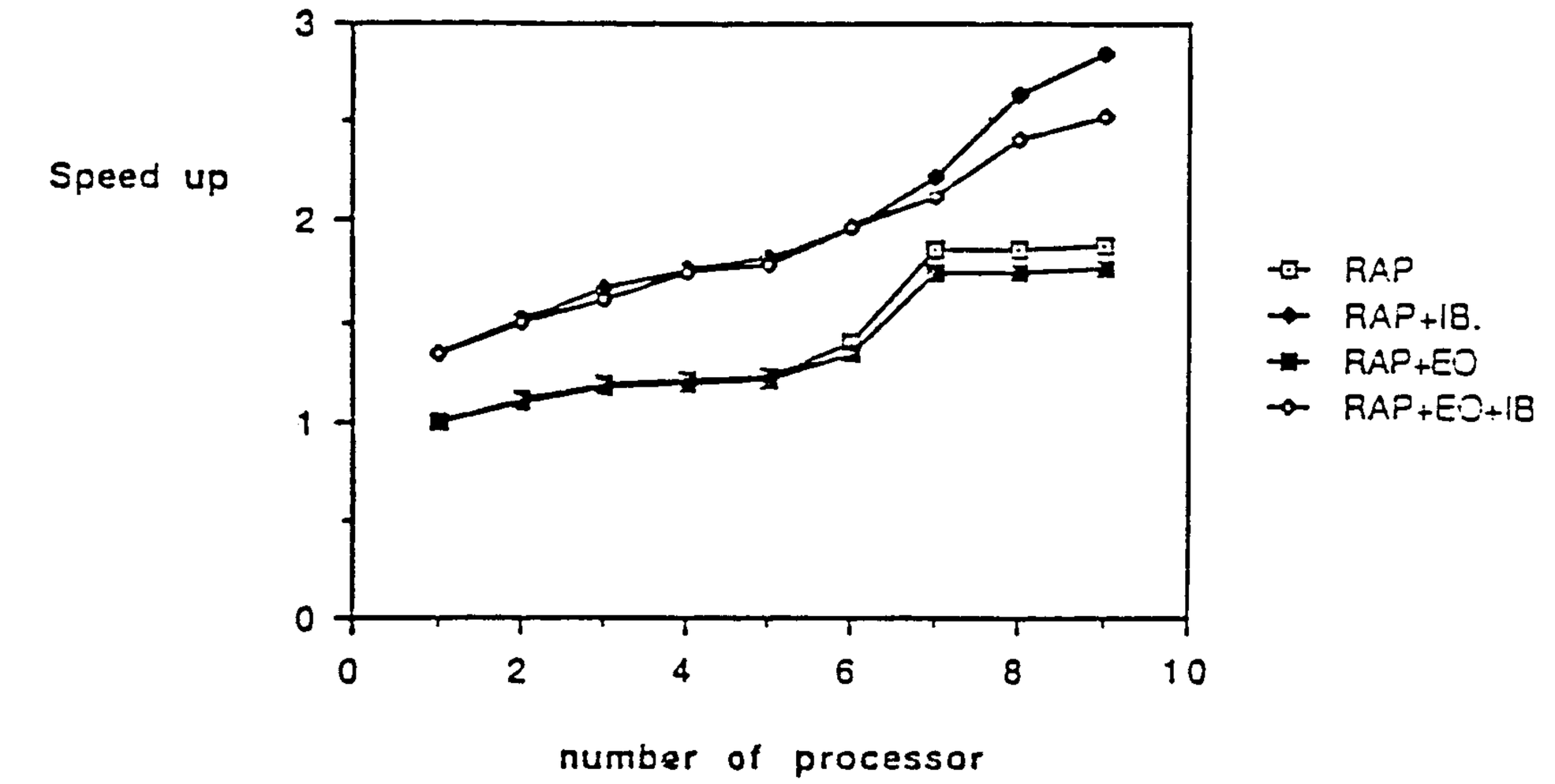



<그림 2.8> check 프로그램에 대한 수행 속도 그래프(times20)

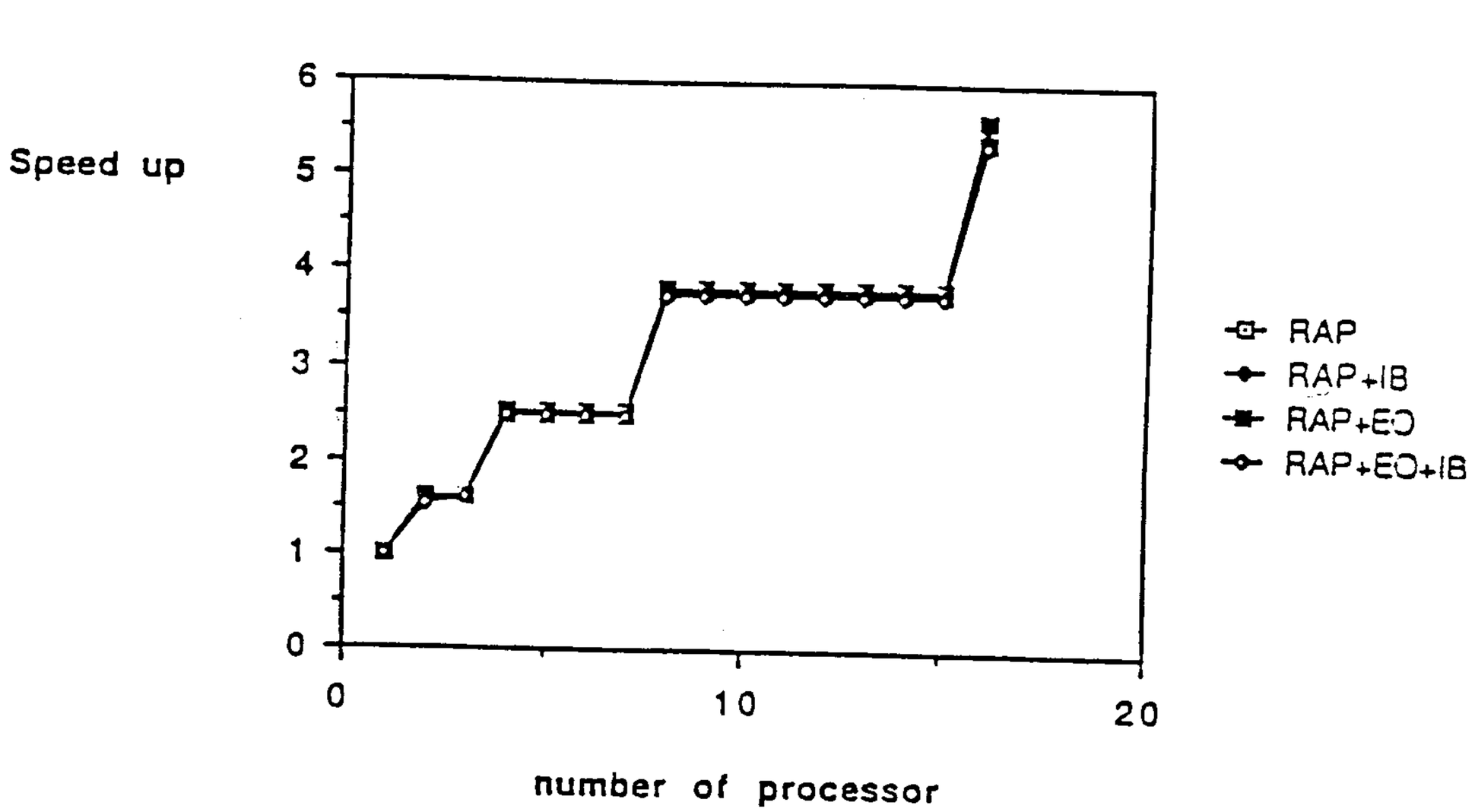



<그림 2.9> check 프로그램에 대한 수행 속도 그래프(times10)

<그림 2.10> arch 프로그램에 대한 수행 속도 그래프




<그림 2.11> map coloring 프로그램에 대한 수행속도 그래프




<그림 2.12> fibonacci 프로그램에 대한 수행 속도 그래프

| Program Name | Number of Processors | First Solution | | | All Solutions | | | Speed-up | |
|---|---|---|---|---|---|---|---|---|---|
| | | time | failure | instructions | time | failure | instructions | first | all |
| eight queen | 1 | 11168740 | 51784 | 2224040 | 1641055168 | 758078 | 32443559 | | |
| | 16 | 907480 | 4182 | 179226 | 12201616 | 56096 | 2393639 | 12.3 | 13.4 |
| map color 1 | 1 | 3565 | 74 | 284 | 248983 | 5688 | 17324 | | |
| | 16 | 1393 | 24 | 139 | 20810 | 474 | 1463 | 2.6 | 12.0 |
| map color 2 | 1 | 3563 | 74 | 284 | 45278 | 1035 | 3152 | | |
| | 16 | 1978 | 39 | 173 | 15022 | 342 | 1059 | 1.8 | 3.0 |
| map color 3 | 1 | 22960 | 520 | 1611 | 162096 | 3708 | 11204 | | |
| | 16 | 1389 | 23 | 140 | 15023 | 342 | 1059 | 16.5 | 10.8 |
| map color 4 | 1 | 4415 | 95 | 338 | 62637 | 1431 | 4364 | | |
| | 16 | 1821 | 35 | 164 | 20809 | 474 | 1317 | 2.4 | 3.0 |
| data base query | 1 | 21577 | 438 | 1669 | 33452 | 676 | 2562 | | |
| | 16 | 1265 | 21 | 124 | 8618 | 196 | 746 | 17.1 | 3.5 |
| symbolic derivation | 1 | 516 | 3 | 81 | 3960 | 41 | 520 | | |
| | 16 | 161 | 0 | 31 | 184 | 1 | 31 | 3.1 | 3.9 |
| quick sort q(X) | 1 | 28598 | 115 | 5262 | 30876 | 171 | 5427 | | |
| | 16 | 28578 | 115 | 5258 | 30856 | 171 | 5423 | 1.0 | 1.0 |

The query of map color 1 is color(A,B,C,D,E).
The query of map color 2 is color(yellow,A,B,C,yellow).
The query of map color 3 is color(A,B,C,D,red).
The query of map color 4 is color(blue,A,B,C,D).

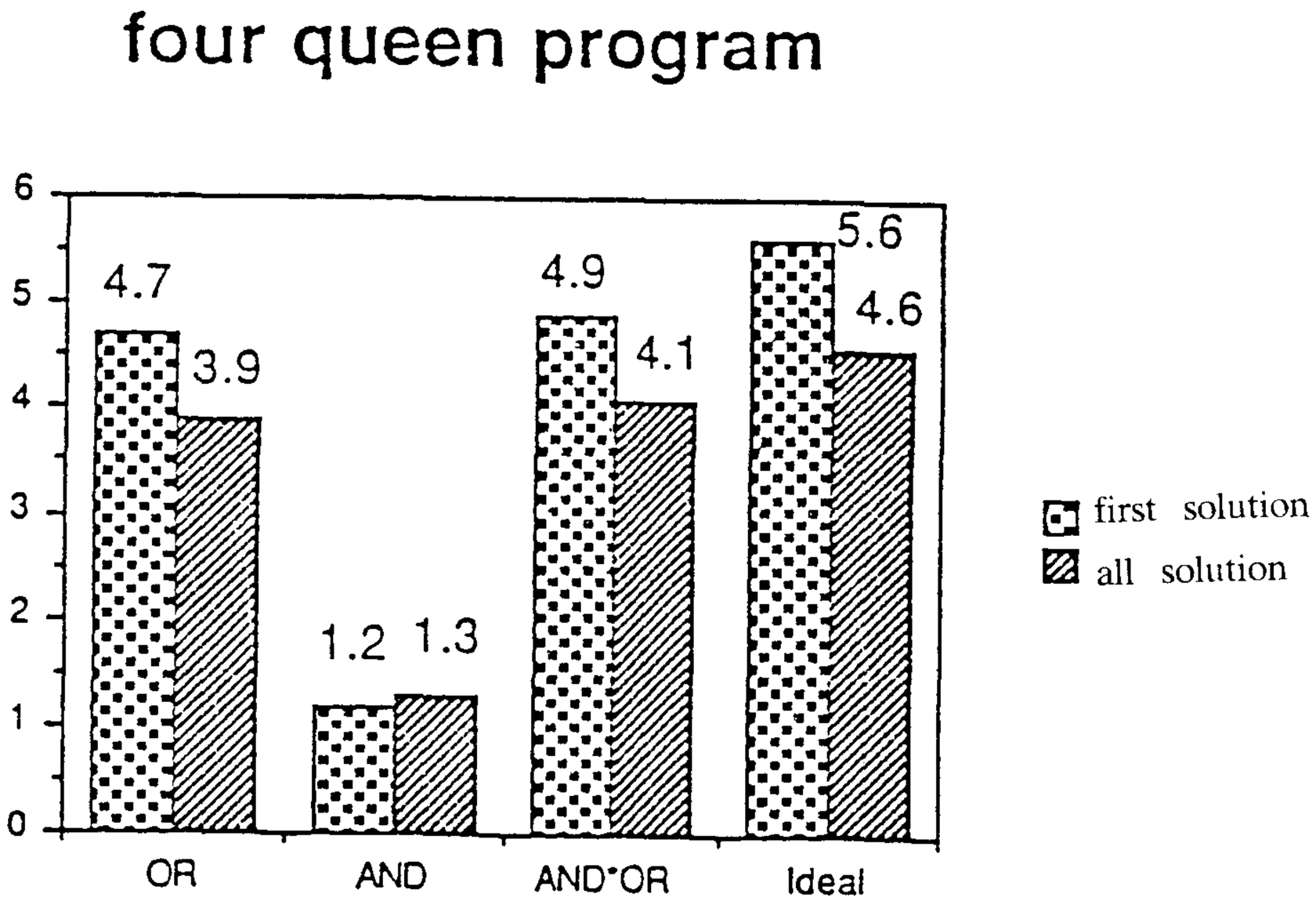
<그림 2.13> OR 병렬 수행을 하였을 때의 성능 향상

### 2.5.3 AND 병렬 수행과 OR 병렬 수행을 동시에 했을 때의 성능

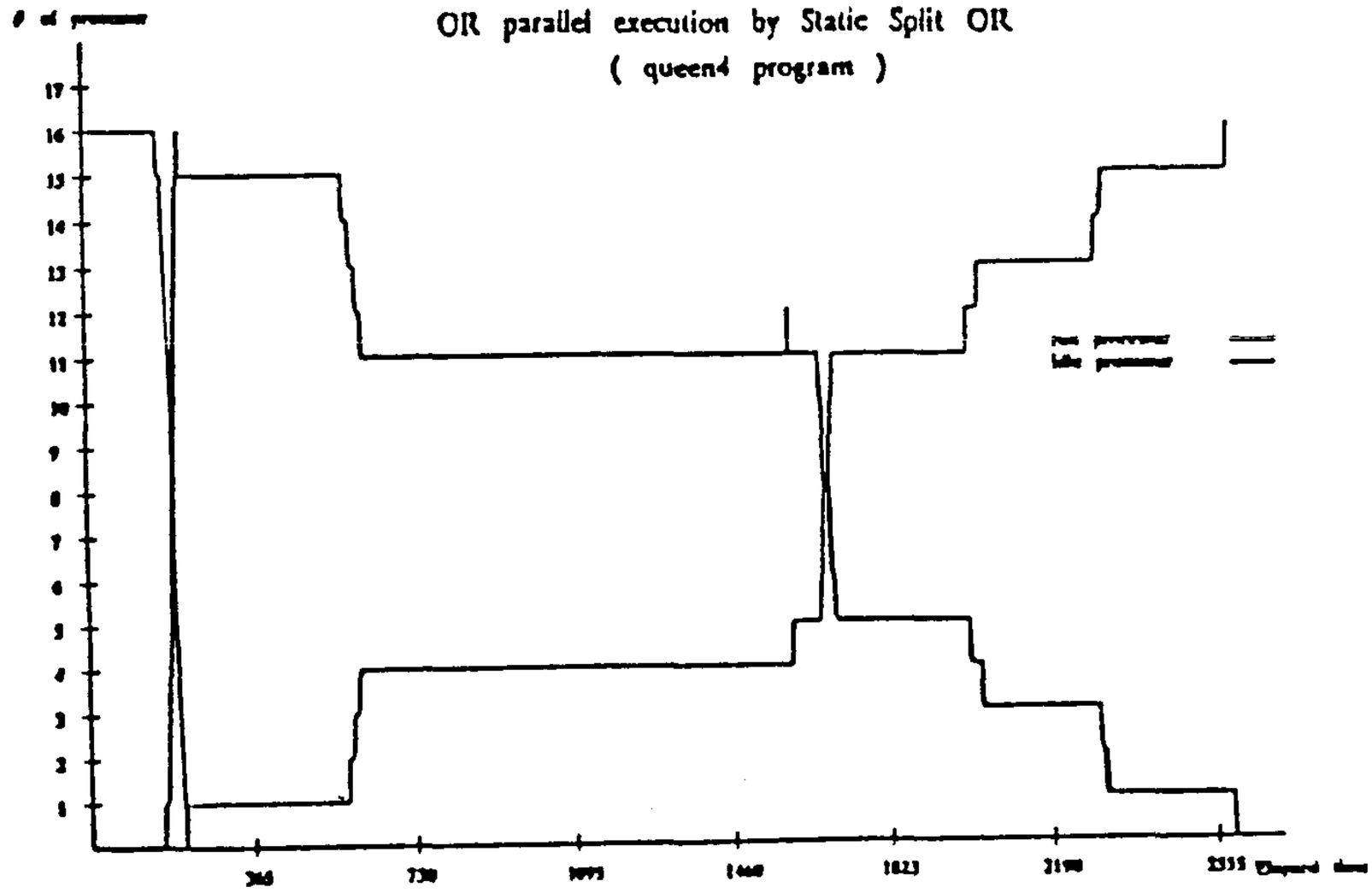
일반적으로, 비결정적인 프로그램은 AND 병렬성과 OR 병렬성을 모두 가지고 있기 때문에, 벤치마크 프로그램으로 비 결정적인 프로그램을 많이 사용하고 있다.

four queen 프로그램은 대표적인 생성-검사(generate-and-test) 형태의 프로그램이며, 이 프로그램을 16개의 번치로 분할하였을 때의 성능이 <그림 2.14>에 나와 있다. 이상적인 경우의 성능은 OR 병렬 수행을 하였을 때의 성능과 AND 병렬 수행을 하였을 때의 성능의 곱이다. 얻어진 성능은 첫번째 해를 구하는 경우에는 4.9이며, 모든 해를 구하는 경우에는 4.1이다. 첫번째 해를 구하는 경우에는 이상적인 경우 성능의 86% 정도인데, 그 이유는 AND 병렬 수행을 시작하는데 전체 수행시간의 10% 정도를 필요로 하기 때문이다. 모든 해를 구하는 경우의 성능은 이상적인 경우 성능의 89% 정도인데, 그 이유는 시간이 지남에 따라서 AND 병렬 고울을 수행하는데 참가하는 프로세서가 많아지기 때문이다.

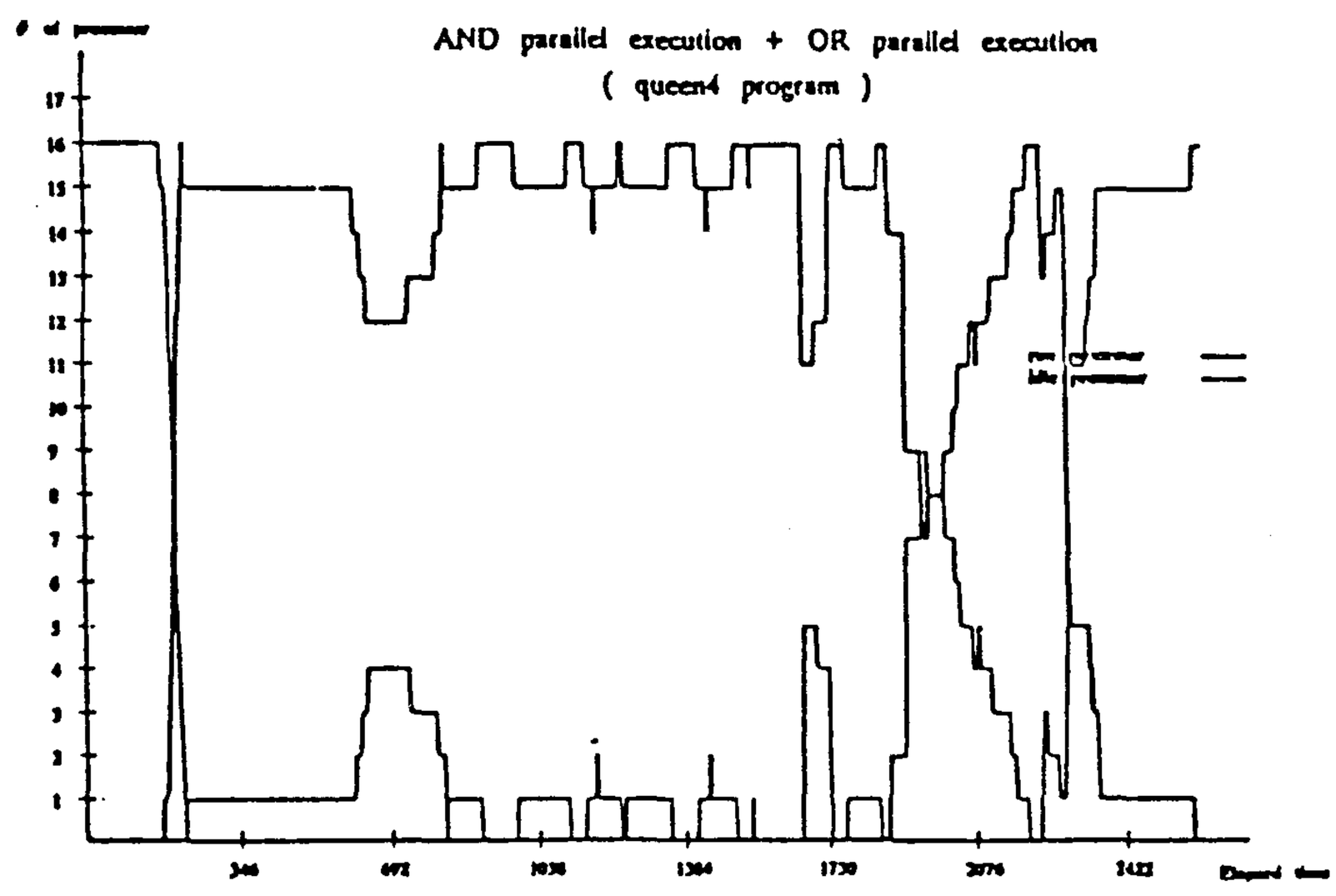
X-WAM에서 four queen 프로그램을 수행시켰을 때의 프로세서 이용도(utilization)를 <그림 2.15>와 <그림 2.16>에 나타내었다. 이 두 그림에서 쉬고 있는 프로세서의 갯수를 비교하면 X-WAM의 프로세서가 AND*OR 병렬 수행을 할 때 훨씬 더 이용도가 높음을 알 수 있다. 하지만 AND 병렬 수행을 할 때의 속도 향상이 작기 때문에 전체적인 성능 향상은 거의 일어나지 않는다. 다른 프로그램에 대한 성능 향상 그래프는 <그림 2.17>, <그림 2.18>, 그리고 <그림 2.19>에 나타내었다. 이 그림들을 통하여 AND*OR 병렬 수행을 하는 경우가 어떤 하나의 병렬성만을 추구하는 경우보다 좋은 성능을 나타낸다는 것을 알 수 있다.



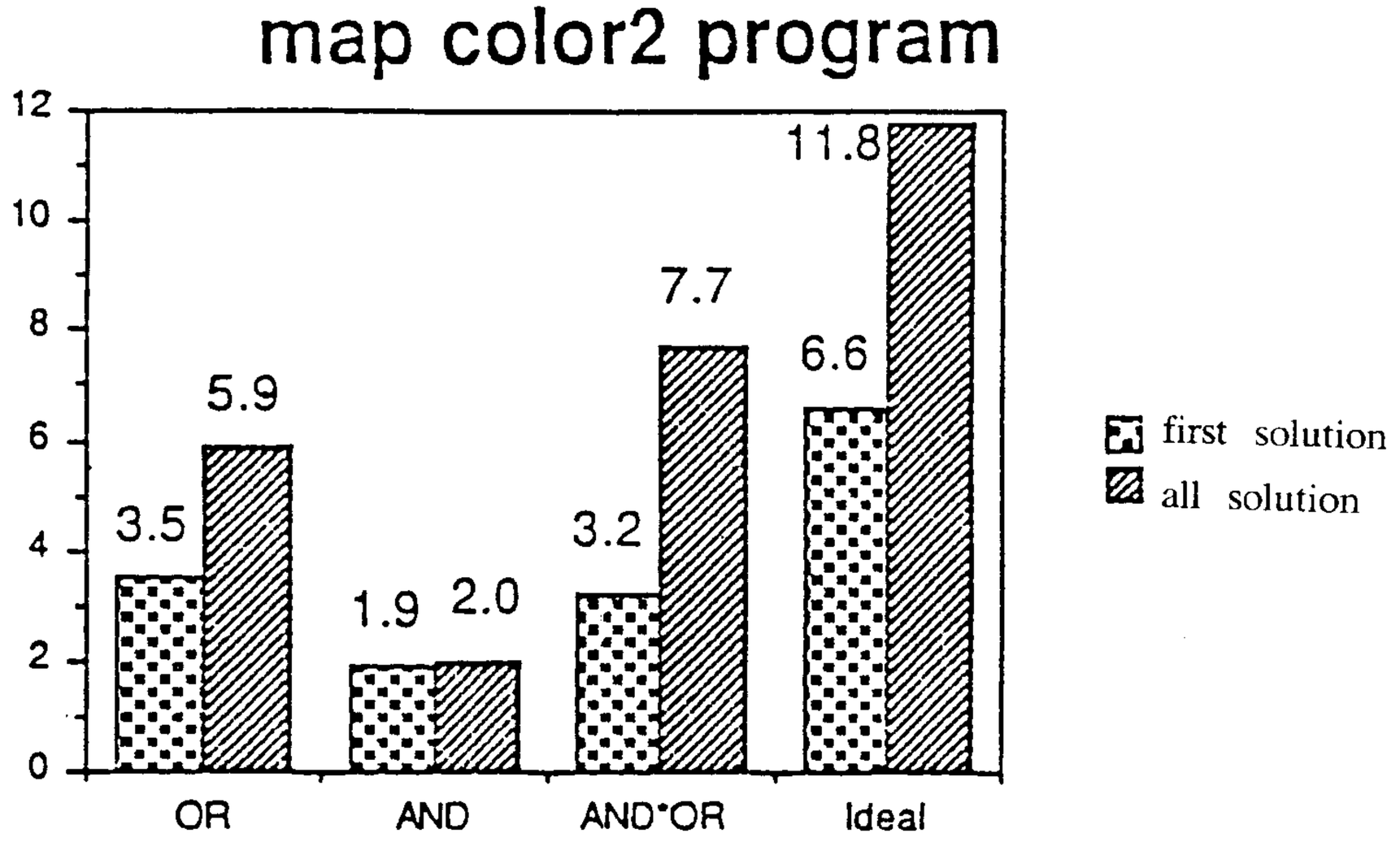

<그림 2.14> AND*OR 병렬 수행 방법의 속도 향상



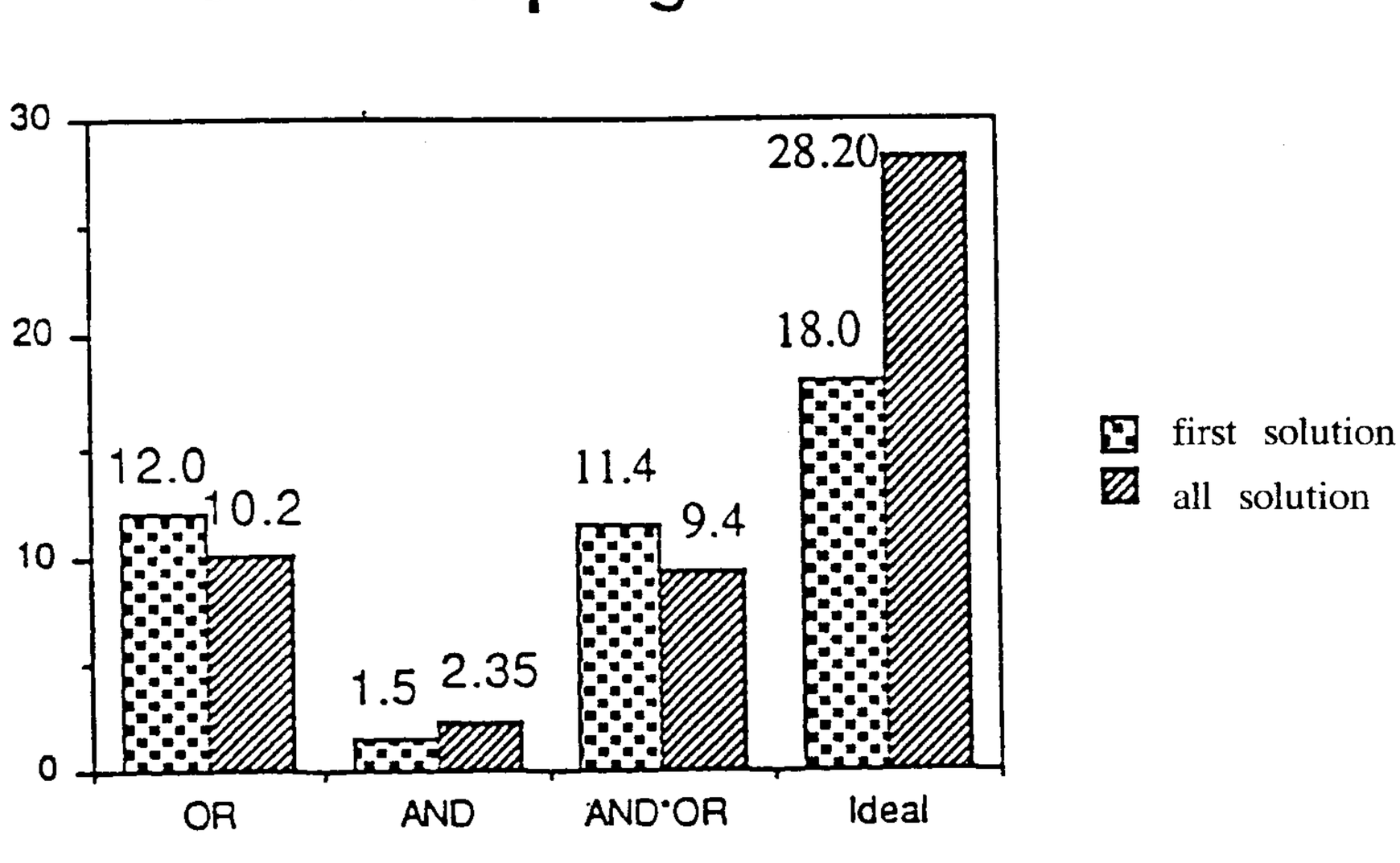

<그림 2.15> OR 병렬 수행을 할때의 프로세서 이용도



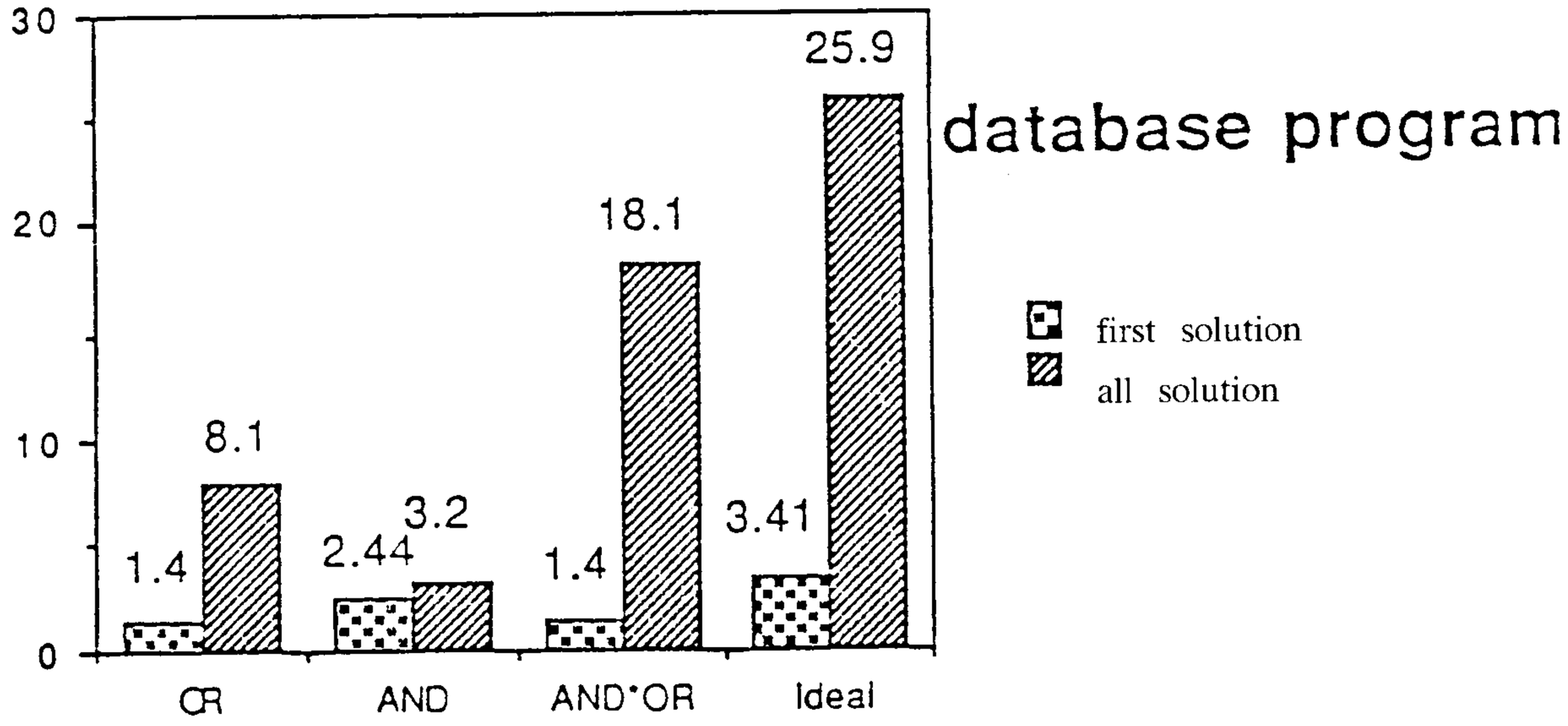

<그림 2.16> AND*OR 병렬 수행에서의 프로세서 이용도




<그림 2.17> AND*OR 병렬 수행 방법의 속도 향상




<그림 2.18> AND*OR 병렬 수행 방법의 속도 향상




<그림 2.19> AND*OR 병렬 수행 방법의 속도 향상

## 2.6 요약

본 장에서는 Prolog 프로그램을 AND*OR 병렬 수행하기 위한 다중 처리기 시스템인 X-WAM을 소개 하였다. X-WAM은 AND 병렬성을 추구하는 경우의 성능과 OR 병렬성을 추구하는 경우의 성능의 곱의 성능을 많은 오버헤드 없이 추구할 수 있는 해결책을 제시하였다. 일반적으로 OR 병렬성은 컴파일시에 쉽게 찾을 수 있는 반면에, AND 병렬성은 컴파일시에 쉽게 찾을 수 없다. 또한 정적 분할 방법은 OR 병렬 수행 모델에서 필요한 수행시 오버헤드를 줄일 수 있는 방법이며, 결과적으로 OR 병렬 수행의 정적 결정과 AND 병렬 수행의 동적 결정이 자연스럽고 타당한 병렬 수행 방법일 것이다.

# 제 3 장 X-WAM 시스템의 구현

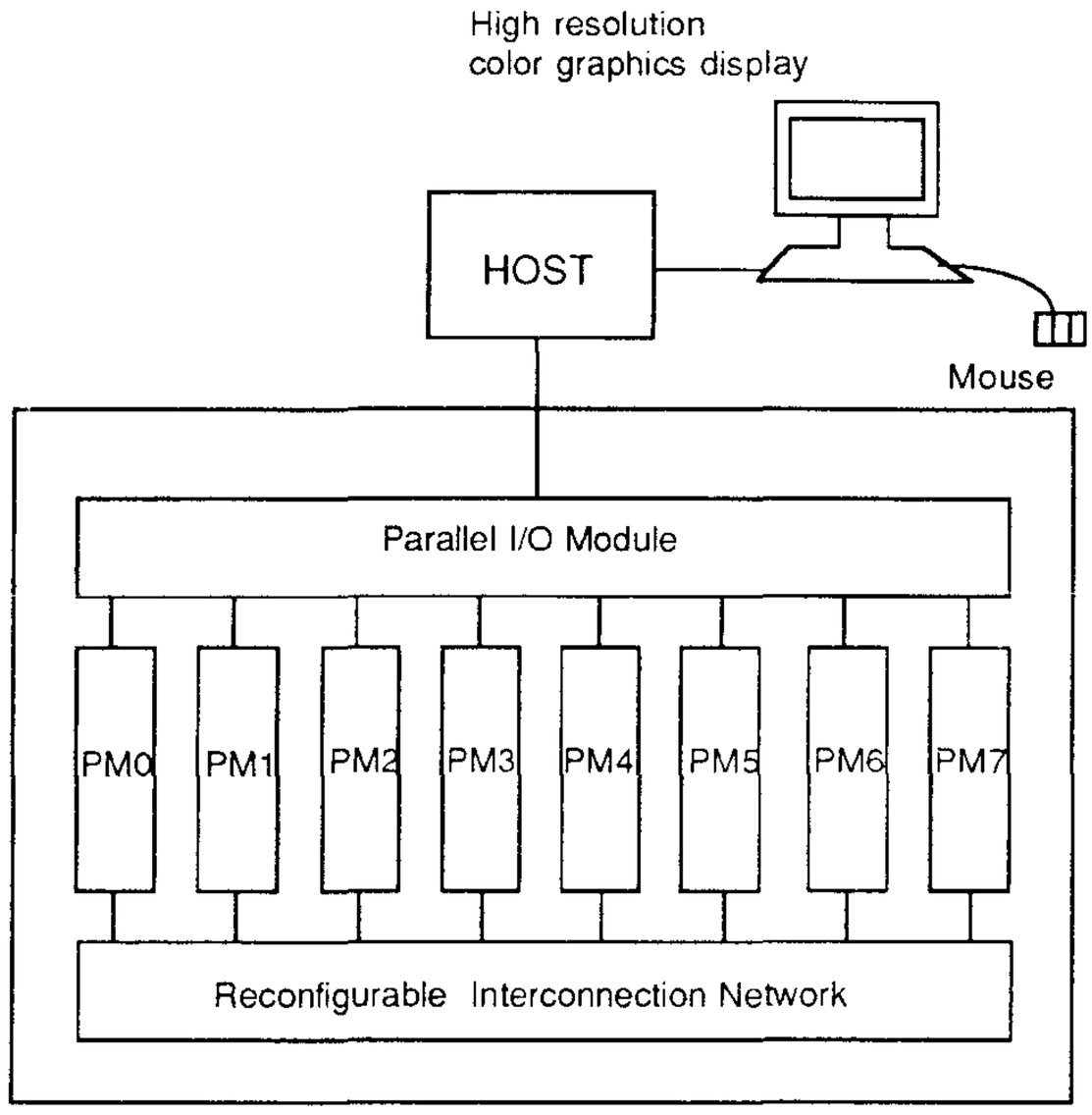
본 보고서에서 제안한 Prolog를 위한 병렬 수행 모델인 X-WAM은 Prolog의 AND 병렬성과 OR 병렬성을 모두 추구하는 모델이다. 하지만 현재 구현된 X-WAM 시스템은 OR 병렬성만을 추구하는 X-WAM-II 모델에 근거를 두고 있다. 그러나 X-WAM의 프로세서 모듈은 AND 병렬성을 추구하기 위한 메시지 전송 기능을 하드웨어적으로 제공하고 있기 때문에 현재의 모듈 모니터에 메시지 처리 루틴을 첨가하면 쉽게 AND 병렬성도 추구할 수 있을 것이다.

본 장에서는 현재 구현한 X-WAM 프로토타입 시스템의 전체 구성과 각 부분에 대하여 기술한다. X-WAM의 하드웨어 시스템은 8 개의 프로세서 모듈과 병렬 입출력 모듈로 구성된 back-end 머신과, 이들의 수행을 제어하면서 사용자와의 인터페이스를 제공하는 호스트로 이루어져 있으며, 소프트웨어 시스템은 프로세서 모듈의 모듈 모니터 프로그램과 호스트에서 수행되는 X-WAM 매니저 프로그램으로 이루어져 있다.

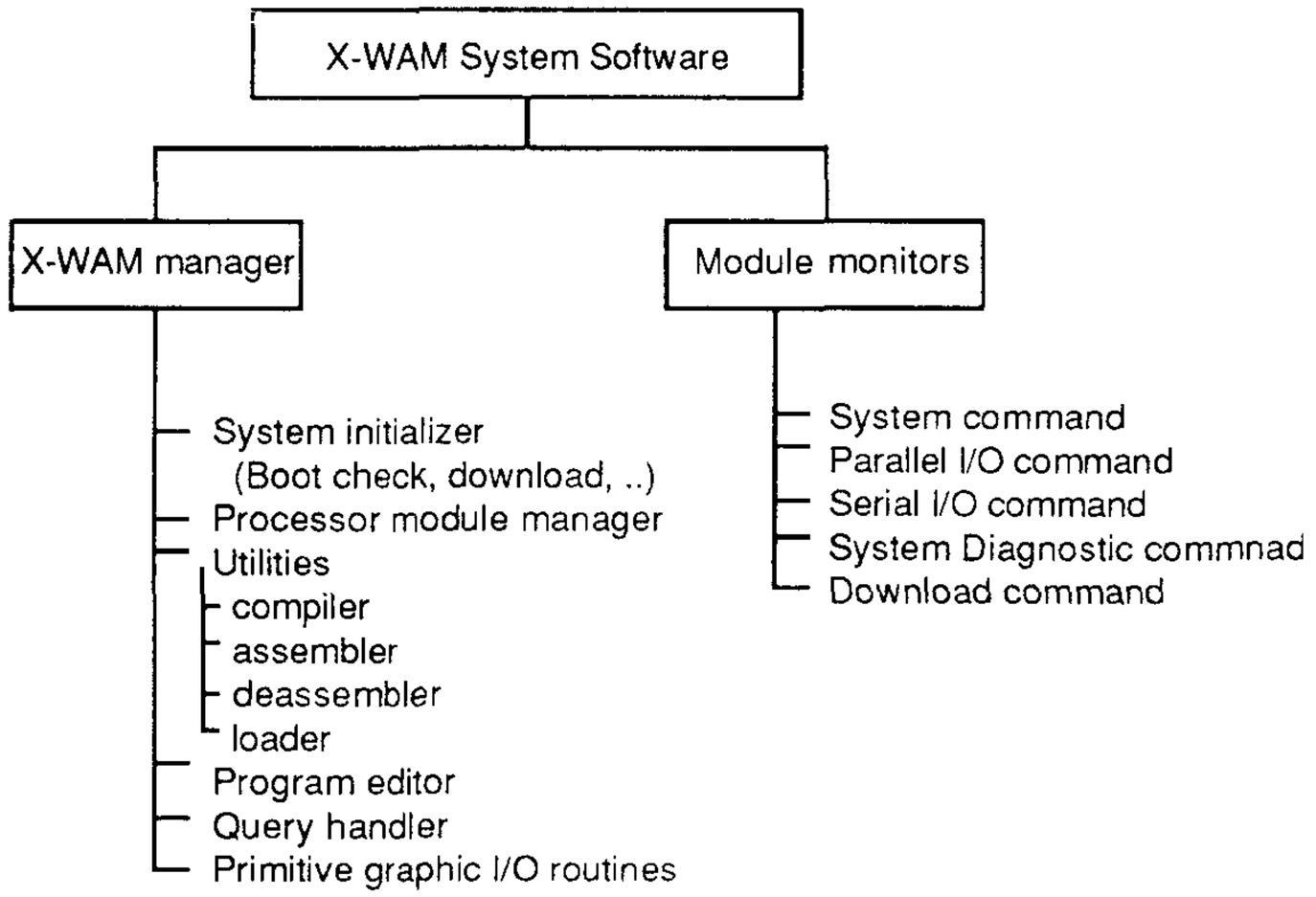
본 장은 다음과 같이 구성된다. 3.1절에서는 X-WAM 프로토타입 시스템의 전체 구성에 대하여 설명하고, 3.2절에서는 X-WAM의 기본 프로세서 모듈과 병렬 입출력 모듈에 대하여 설명한다. 3.3절에서는 여러 시스템 소프트웨어들 (X-WAM 매니저와 모듈 모니터)에 대하여 설명하며, 3.4절에서는 X-WAM의 소프트웨어 개발 환경(사용자 인터페이스와 여러 유틸리티들)에 대하여 설명한다.

## 3.1 프로토타입 시스템의 구성

<그림 3.1>에 X-WAM의 하드웨어 구성을 나타내었다. 호스트 컴퓨터로는 386 PC (25 MHz)를 사용하며 컬러 그래픽스 디스플레이와 마우스가 장착되어 있다. 호스트 컴퓨터는 시스템 제어 및 사용자 인터페이스를 담당하며 병렬 입출력 모듈을 통하여 각 프로세서 모듈과 통신한다. 병렬 입출력 모듈에는 각 프로세서 모듈이 연결되어 있으며 프로세서 모듈 간의 통신은 소프트웨어에 의하여 호스트를 통하거나 재 구성이 가능한 상호 연결 네트워크(Interconnection Network)를 통해서 할 수 있다.




<그림 3.1> X-WAM의 하드웨어 구성도

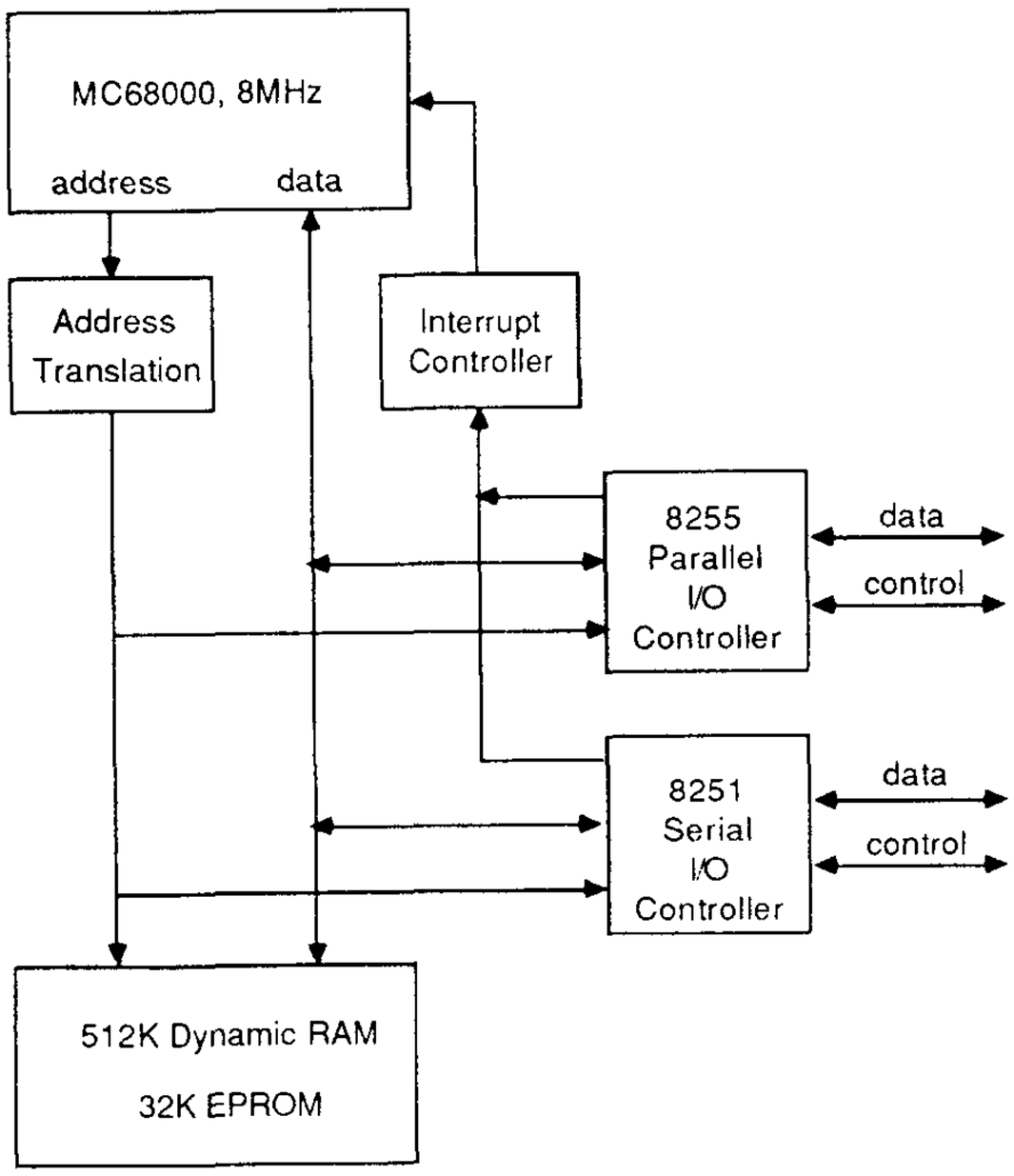



<그림 3.2> X-WAM의 소프트웨어 구성도

<그림 3.2>에 X-WAM의 소프트웨어 구성을 나타내었다. 소프트웨어 시스템은 프로세서 모듈의 모듈 모니터 프로그램과 호스트에서 수행되는 X-WAM 매니저 프로그램으로 이루어져 있다. 모듈 모니터 프로그램은 프로세서 모듈 내의 ROM에 내장되며, 시스템 제어, 시스템 진단, 병렬/직렬 입출력, 프로그램 다운로우드 등의 기능을 제공한다. X-WAM 매니저 프로그램은 시스템 관리, 프로세서 모듈 관리, 사용자 인터페이스, 그리고 소프트웨어 개발 환경을 제공한다.

## 3.2 프로세서 모듈과 병렬 입출력 모듈

X-WAM의 back-end 머신의 프로세서 모듈의 CPU로는 모토롤러의 16 비트 마이크로 프로세서인 68000을 채택하였으며 각 프로세서 모듈의 메모리는 모니터 프로그램이 내장되어있는 32 KByte의 ROM과 X-WAM 에뮬레이터, 사용자 프로그램 등을 저장하기 위한 512 KByte의 RAM으로 이루어져 있다.



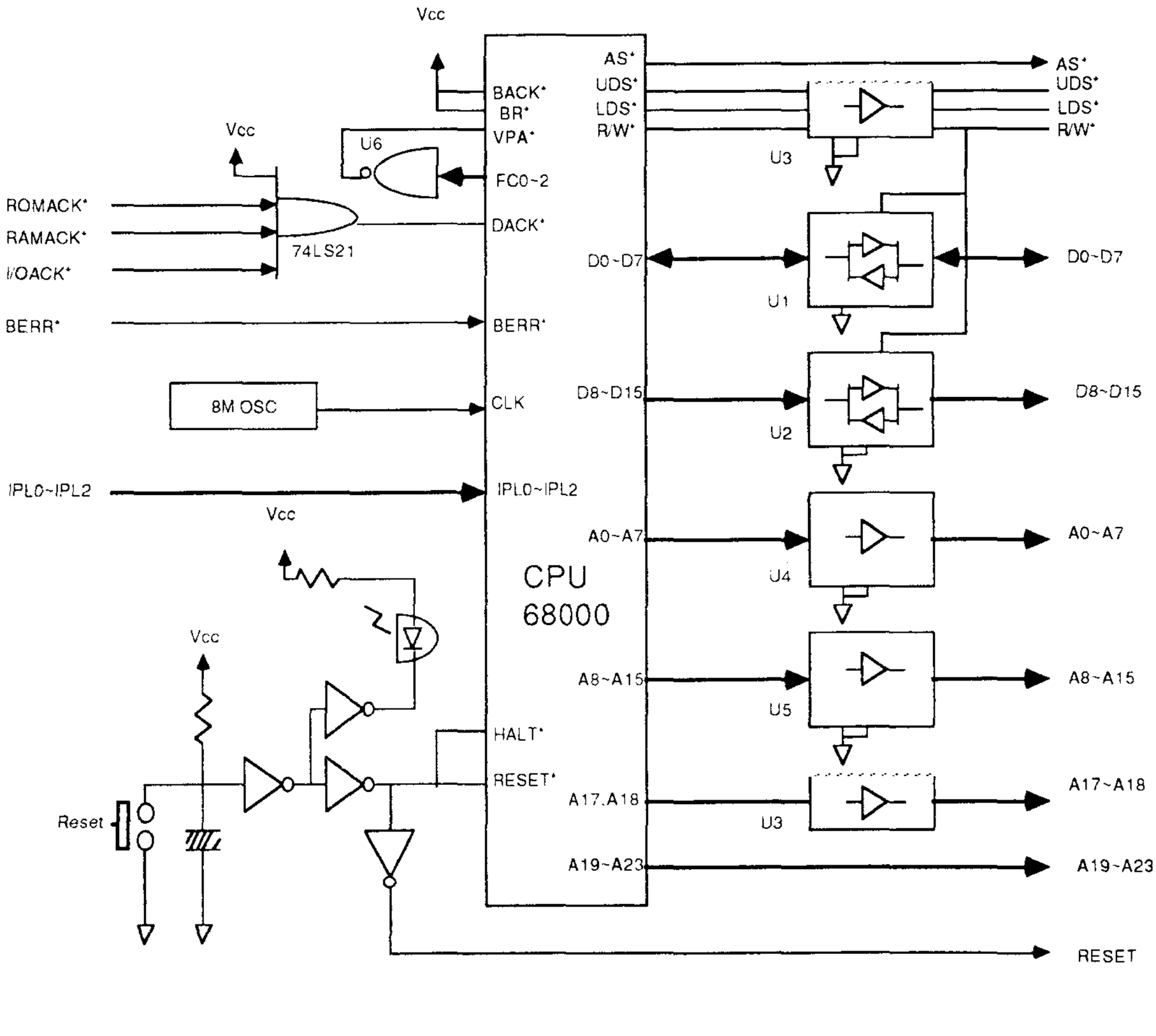

<그림 3.3> 프로세서 모듈의 블럭 다이어그램

또 각 프로세서 모듈에는 호스트와의 통신을 위한 병렬 입출력 포트 및 프로세서 모듈 간의 통신을 위한 4개의 직렬 입출력 포트가 설치되어 있다. 병렬 입출력 포트의 컨트롤러로는 8255를, 직렬 입출력 포트의 컨트롤러로는 8251를 사용하였다. 직렬 입출력 포트는 X-WAM의 AND 병렬성 구현 및 메시지 전송에 기반을 둔 범용 다중 처리기 시스템으로의 확장 가능성을 고려하여 설계한 것이며, 시스템 진단 및 디버깅 용으로 사용할 수도 있다. <그림 3.3>에 프로세서 모듈의 블럭 다이어그램을 나타내었다.

### 3.2.1 CPU 회로

16 비트 마이크로 프로세서인 MC 68000을 중심으로 한 CPU 회로는 프로세서 모듈내의 각 부분에 필요한 제어신호를 공급하고 프로그램의 sequencing을 담당하는 중추적인 회로이다.

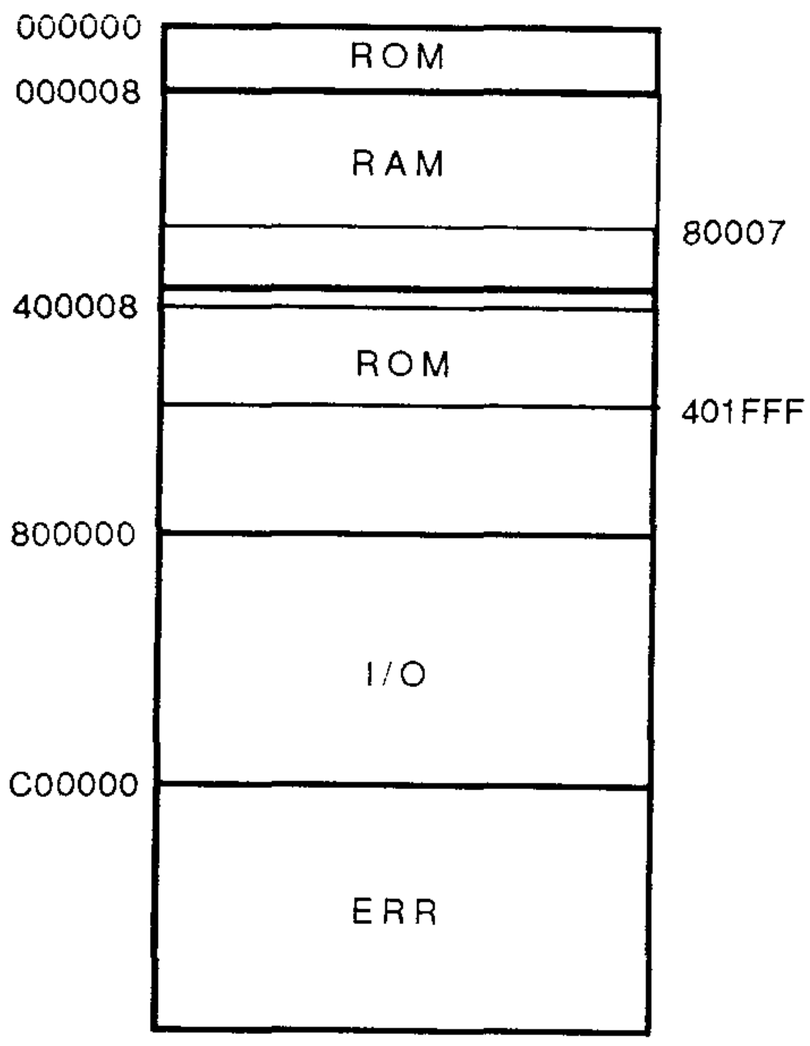
u1,u2는 16비트 데이타버스의 양방향성 버퍼이며 R/W*신호에 의해 방향이 제어된다. u4, u5, 그리고 u3의 일부는 어드레스 버스 버퍼이며 A1부터 A18까지를 버퍼링한다. A19부터 A23까지의 어드레스신호는 디코더 한 곳에만 연결되므로 버퍼를 쓰지 않았다. u3의 나머지 단위버퍼들은 다른 제어신호를 중폭하는데 사용된다. 모든 버퍼출력은 10개의 TTL입력을 drive할 수 있다. 68000의 모든 메모리 액세스는 DTACK* 입력선으로 응답이 이루어져야 실행이 계속되는데 74LS21 4입력 AND 게이트는 각 부분에서 들어오는 응답신호들을 하나로 모으는 역할을 한다. CPU의 시스템 클럭은 8MHz이며 8M OSC에서 곧바로 CPU로 공급된다. IPL0*, IPL1*, IPL2* 입력신호선은 I/O part로부터 인터럽트신호를 받아들인다. 인터럽트가 없을 때 이 신호선들은 모두 high상태를 유지하고 있으며 이 상태에서 모두 일곱개의 각기 다른 우선순위의 인터럽트를 받아들일 수 있다. u6은 입출력 포트에 의해 인터럽트가 발생하였을 때 인터럽트 벡터를 응답해주지 않아도 되는 자동 인터럽트(auto-interrupt)로 구성하기 위한 것이며, 이 때 각 인터럽트는 그 우선순위에 따라 벡터가 결정된다. RESET*와 HALT* 입력은 RESET 스위치를 눌렀을 때 또는 전원이 처음 공급될 때 콘덴서의 과도상태에 의하여 활성화된다. Reset신호는 CPU에 공급되는 외에도 LED를 켜거나 I/O part의 PIO를 초기화 시키는 데에 사용된다. <그림 3.4>은 CPU 회로의 구성을 나타낸 것이다.




<그림 3.4> CPU 회로의 구성도

### 3.2.2 Decoder 회로

Decoder 회로는 ROM, RAM, I/O등 각 모듈에 어드레스 공간을 할당하는 역할을 한다. 할당된 메모리 맵은 <그림 3.5>와 같다.

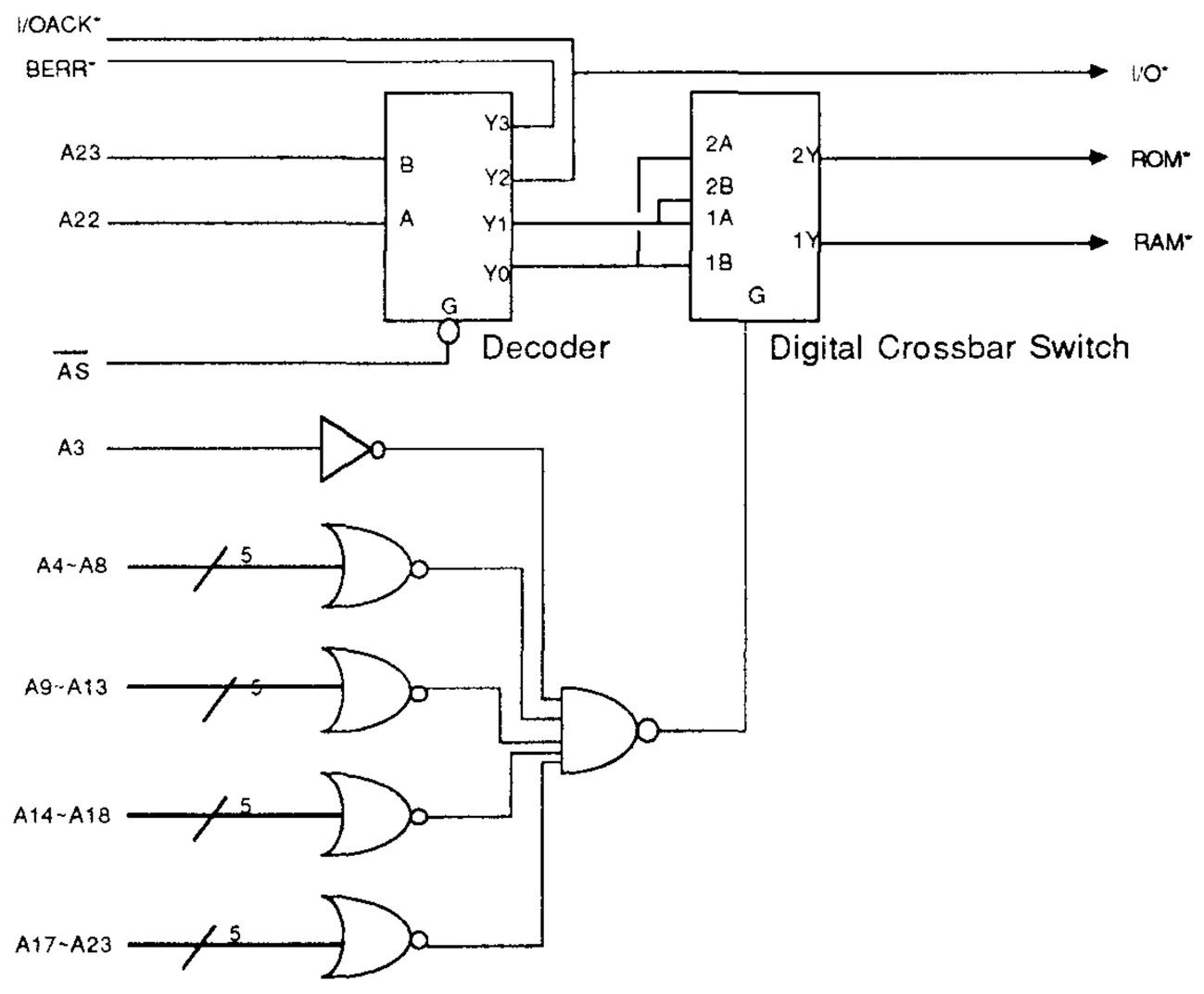



<그림 3.5> 메모리 맵

RAM, ROM, I/O, ERR의 순서로 16M 바이트의 주소영역이 4등분된다. ERR는 사용하지 않는 공간이며 이 영역이 액세스되면 에러로 간주하여 인터럽트를 발생시킨다.

CPU 68000은 reset신호에 의하여 처음 초기화되었을때 0번지에서부터 엑서스를 시작하며, 4~7번지에 있는 데이타를 초기번지로 하여 프로그램 실행을 시작한다. 이 기능은 프로세서의 하드웨어에 의해 결정되어 있어서 바꿀 수 없고 그 내용이 고정되어 있어야 하므로 이 address 0~7은 ROM이 할당되어야 한다. 그런데 8번지부터 255번지까지는 인터럽트 벡터 테이블에 보관되며 프로그램에 의해 동적으로 수정되어야 하므로 RAM이 할당되어야 한다. 이와 같은 이유로 0~7번지영역은 <그림 3.6>과 같은 특별한 회로를 통해 ROM영역으로 분리된다.

디코더 74LS139의 Y0과 Y1은 4개의 4등분 영역중 첫 번째와 두 번째 영역일 때 각각 활성화되며 두 개의 2-to-1 selector로 구성된 디지탈 크로스바 스위치로 연결되어 G입력에 따라 1Y와 2Y의 출력에 연결되는 두 입력신호가 서로 바뀌게 된다. 한편 A3 이상의 모든 번지 신호선은 4개의 5입력 NOR 게이트와 1개의 5입력 NAND, 1개의 인버터로 조합된 21입력 OR 로 연결되며 이 출력은 앞에서 설명한 스위치의 G입력과 연결되어 0~7번지가 액세스될 때만 low상태로 떨어진다. 따라서 이 영역이 액세스되면 첫 4등분 영역의 디코더출력이 ROM으로 바뀌어 위의 조건을 만족시키게 된다. 번지가 할당되지 않은 4번째 4M 바이트영역(0xc00000~0xffffff)안에 있는 번지가 접근되면 자동적으로 BERR* 신호선을 활성화시켜서 CPU에 인터럽트를 발생시킨다.



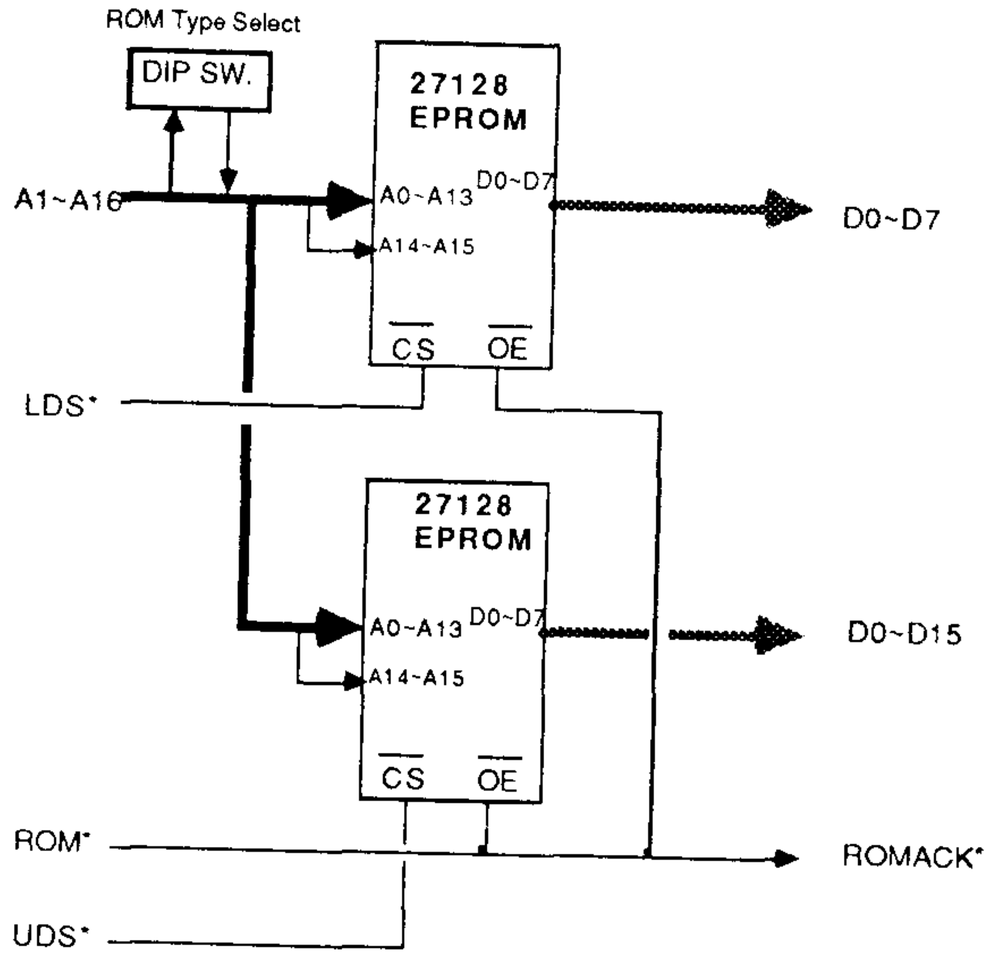

<그림 3.6> Decoder 회로의 구성

### 3.2.3 ROM 회로

ROM 회로는 응용 프로그램을 적재하여 실행시키거나 호스트와의 입출력을 맡게되는 모듈 모니터 프로그램을 저장하는 곳이며 두 개의 16K 바이트 EPROM (27128)을 이용하여 32K 바이트의 용량을 갖는다.

ROM의 액세스 시간은 AS*가 액세스되는 주기에 비해 충분히 작기때문에 ROM enable신호가 바로 acknowledge로 응답된다.

ROM의 번지입력중 일부는 DIP스위치와 연결되어 추후에 메모리를 확장시킬때 선택할 수 있게 하였다. 이 스위치의 선택으로 각각의 ROM은 최고 64K바이트까지 확장할 수 있다. <그림 3.7>은 ROM 회로를 나타낸 것이다.




<그림 3.7> ROM 회로

### 3.2.4 DRAM 회로

RAM 회로는 16개의 256K dynamic RAM으로 구성되어 512K 바이트 용량의 메모리를 제공한다. 이 DRAM 회로는 refresh를 위한 회로, refresh/access arbitration회로, 멀티플렉서회로, 메모리 제어신호 발생회로, 그리고 DRAM 회로로 나눌 수 있다.

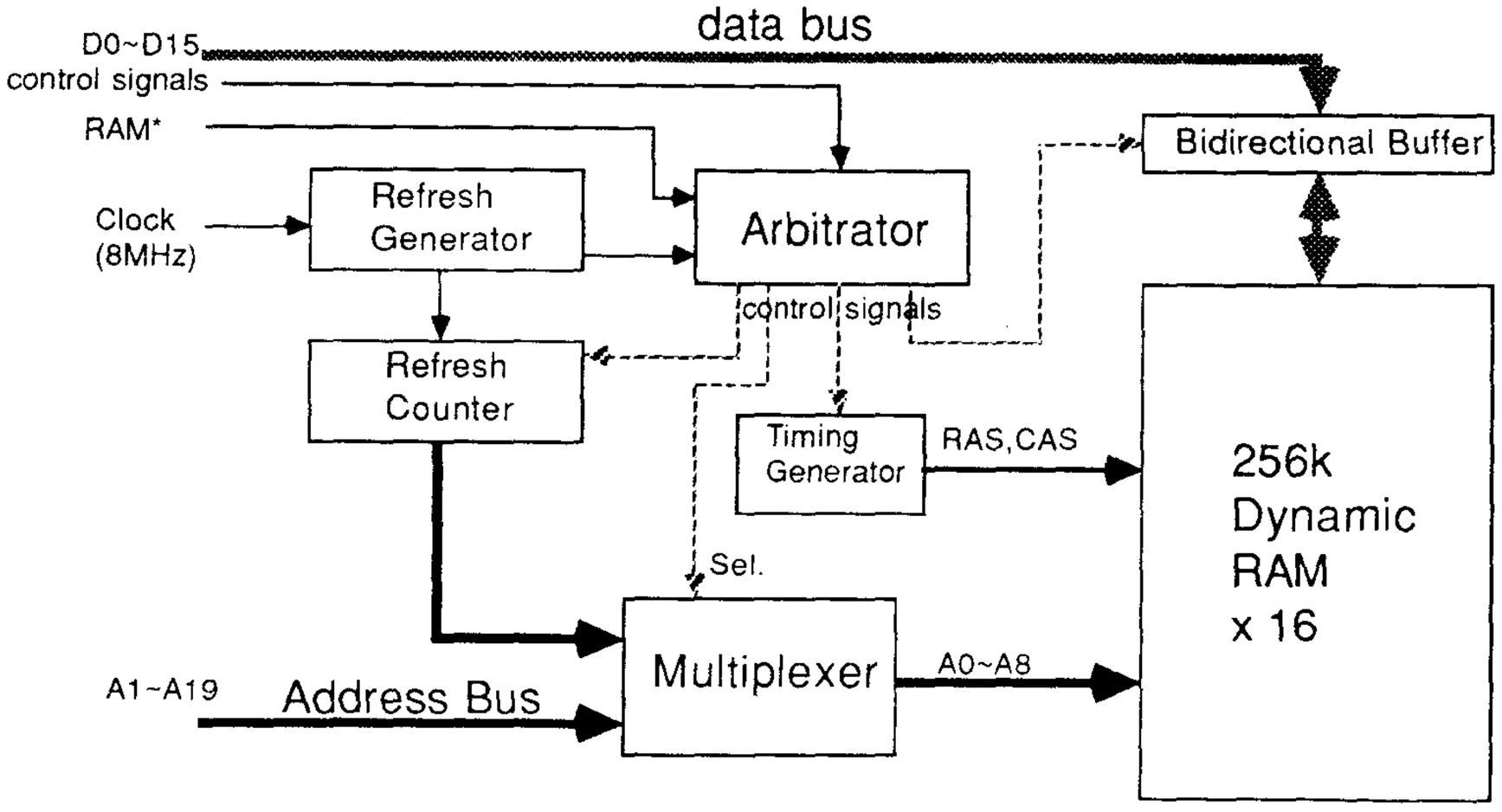
DRAM refresh 회로 : CPU 회로로부터 공급되는 8 MHz의 클럭은 64진 카운터와 연결되어 매 8 마이크로초마다 DRAM refresh신호를 발생시킨다. 발생된 신호는 arbitration회로로 입력되어 CPU로부터 RAM이 액세스되고 있지 않은 시간을 기다려서 refresh 카운터를 1씩 증가시킨다.

Refresh/Access arbitration 회로: RAM이 CPU에 의해 액세스되고 있으면 액세스가 끝날 때까지 refresh회로를 지연시키고 refresh중이면 RAM의 액세스를 지연시킨다. 이 회로는 두개의 D-플립플롭에 의해 '시이소오'동작을 하도록 구성된다. 여기서 만들어진 제어신호들은 refresh회로, 멀티플렉서, 타이밍 발생회로, 그리고 데이타버퍼와 연결된다.

Multiplexer회로: refresh중일 때와 메모리 액세스중일 때 refresh카운터의 출력과 18비트의 어드레스버스를 각각 선택한 후 9 비트로 시분할하여 DRAM의 번지입력으로 공급된다.

RAM제어신호 발생회로: DRAM을 동작시키기 위한 RAS와 CAS신호를 발생시킨다.

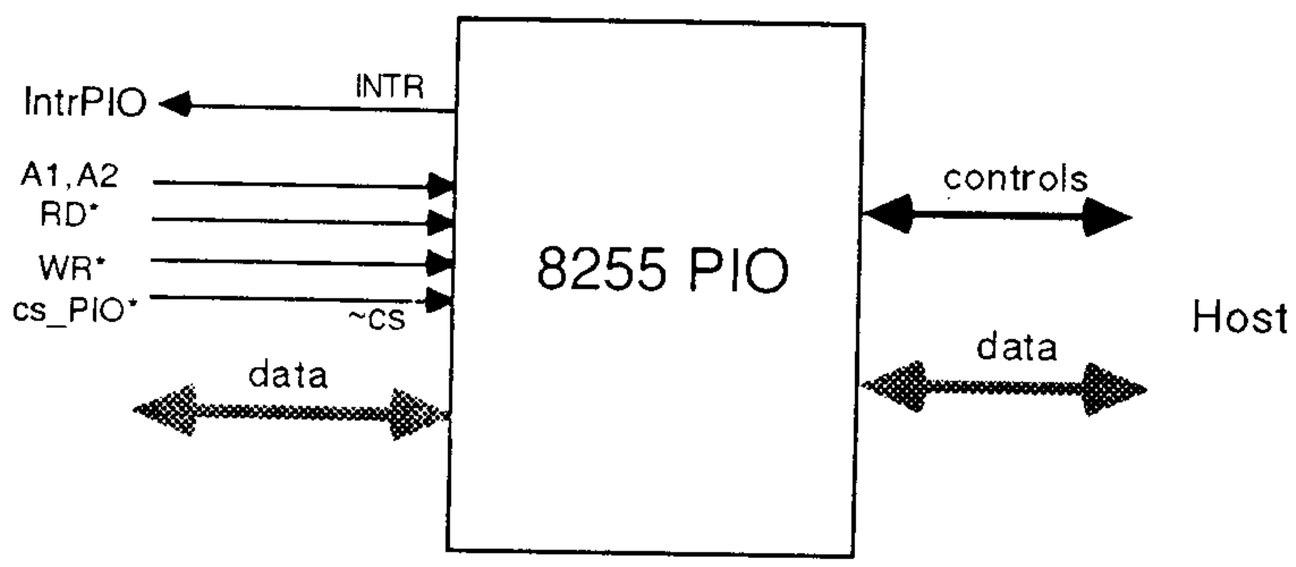
DRAM 회로: 16개의 256k*1 DRAM이 256k*16비트의 용량을 제공한다. 각 DRAM의 입출력은 서로 연결되어 74LS245 양방향성 버퍼를 통해 데이타버스로 전달된다. RAM의 WE* 입력은 8개씩 묶여서 UDS*와 LDS*에 의해 각각 제어된다. 이 외의 모든 핀은 16개씩 공통으로 묶여 사용된다. <그림 3.8>은 DRAM 회로의 구성도이다.




<그림 3.8> DRAM 회로

### 3.2.5 병렬 입출력포트 회로

1개의 8255 PIO는 두개의 8비트 입출력포트를 제공하며 호스트와 연결된다. 이 포트를 통해 호스트로부터 응용 프로그램이 적재되고 프로그램의 입출력이 호스트에 의해 처리된다. 두 개의 포트는 프로그램에 의해서 입출력을 전환할 수 있으나 일반적으로 하나는 입력포트로, 나머지 하나는 출력포트로 사용된다. PIO의 IntrPIO신호선은 인터럽트 우선순위 인코더(priority encoder)에 연결되어 호스트로부터 새로운 데이타가 도착했을 때 인터럽트를 발생시킨다. <그림 3.9>에 PIO의 연결내용을 나타내었다.




<그림 3.9> 병렬 입출력포트

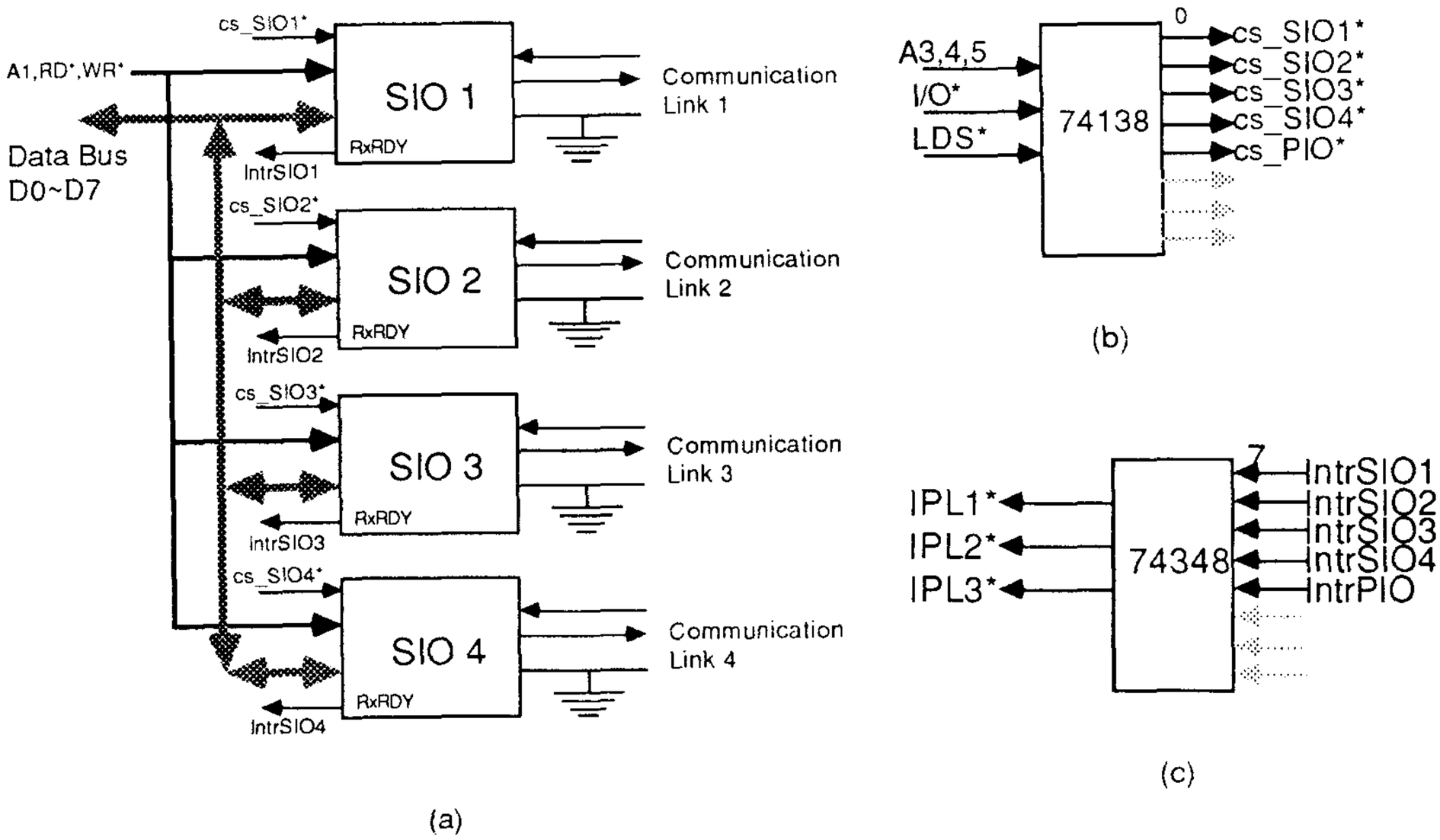
### 3.2.6 직렬 입출력포트 회로

4개의 8251 SIO는 각각이 하나의 입력포트와 하나의 출력포트를 제공하며 프로세서 모듈 사이의 통신에 사용된다. 따라서 각 프로세서 모듈은 4개의 통신 채널을 갖고 다른 프로세서 모듈과 임의의 망을 형성할 수 있다. 직렬 입출력포트 회로에 속하는 회로로는 이 4개의 SIO외에도 각 포트를 선택하는 디코더 회로와 인터럽트의 우선 순위를 엔코딩하는 우선순위 인코더가 있다.

8251 SIO: 한 개의 SIO는 다른 한 개의 프로세서 모듈과 연결되어 통신할 수 있다. 각 SIO가 외부 프로세서 모듈과 연결되는 선은 한 개의 송신선(TXD), 한 개의 수신선(RXD), 그리고 접지선의 3가닥이며 이 송수신선을 통해 비동기 직렬 송수신으로 통신한다. 수신선을 통해 새로운 데이타가 들어오면 SIO의 RxRDY출력이 활성화되어 우선순위 인코더를 통해 인터럽트를 발생시킨다 (<그림 3.10>(a)).

포트 디코더 회로: CPU가 4개의 SIO와 1개의 PIO중의 하나를 선택하게 해주는 회로이며 이 출력은 각 포트의 칩 선택 입력으로 연결된다 (<그림 3.10>(b)).

우선순위 인코더: 4개의 SIO와 1개의 PIO에서 발생된 인터럽트의 우선순위를 결정해주는 회로이며 CPU의 IPL 1,2,3과 연결되어 CPU에 가장 높은 우선순위의 인터럽트를 우선적으로 CPU에 알린다 (<그림 3.10>(c)).

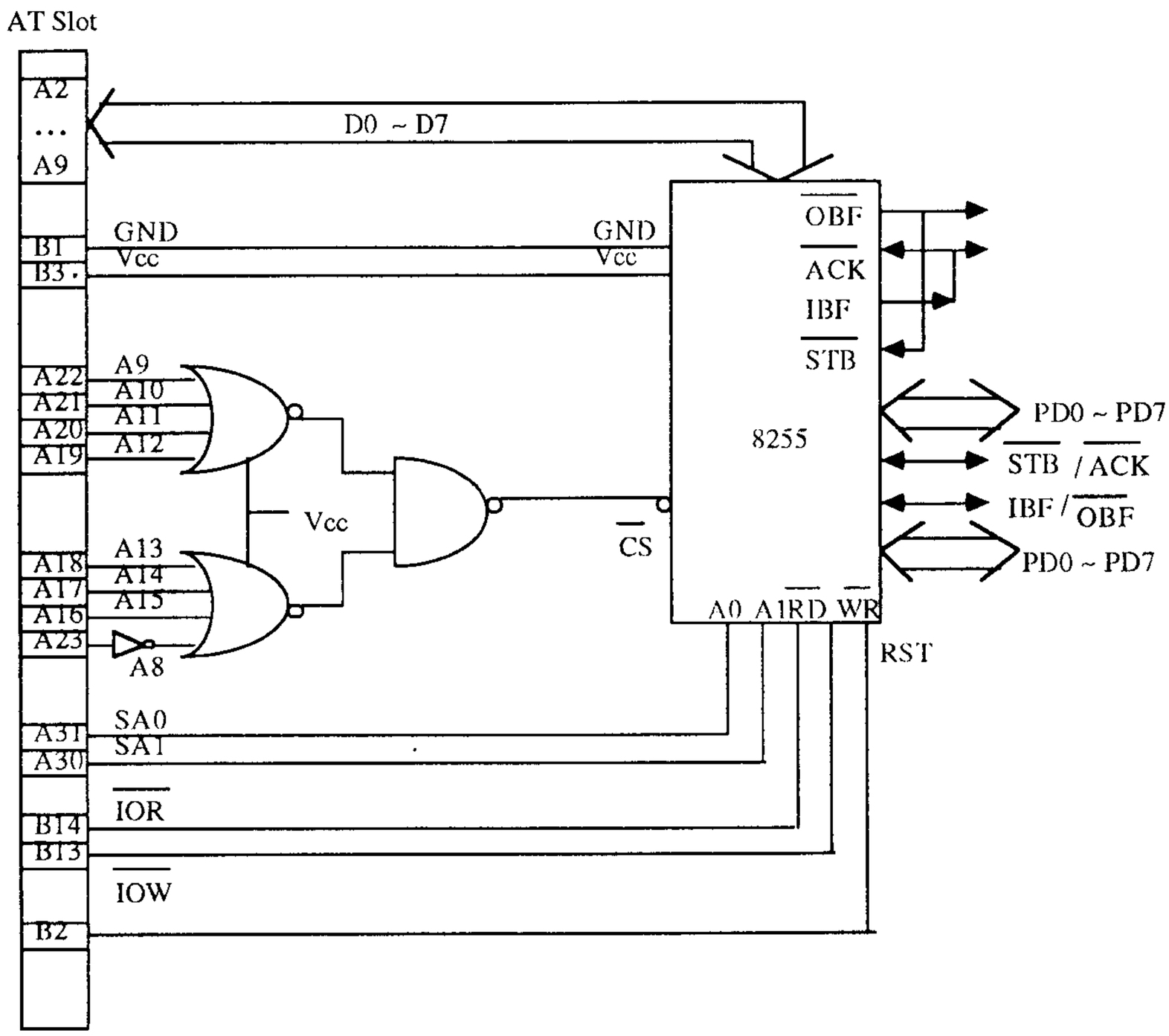



<그림 3.10> 직렬 입출력 회로

### 3.2.7 병렬 입출력 모듈

병렬 입출력 모듈은 호스트와 프로세서 모듈 간의 통신 기능을 제공하기 위한 모듈로서, 각 프로세서 모듈과 병렬 입출력 버스로 직접 연결되어 있다. 호스트의 어드레스를 디코우딩하여 지정된 프로세서 모듈에 있는 8255의 상태 레지스터 또는 데이타 레지스터에 엑세스할 수 있도록 하는 기능을 제공한다.

호스트는 386 PC로서 확장 슬롯에 8255 chip을 호스트와 68000 프로세서간에 handshaking하게 하였다. AT 슬롯에서 $100 ~ $1ff 주소를 디코딩하여 8255 칩을 선택하게 하였다. 앞의 68000 프로세서의 포트간의 handshaking하는 것과 AT와 68000 프로세서간의 포트 handshaking은 동일하다. <그림 3.11>은 병렬 입출력 모듈의 회로를 나타낸 것이다.

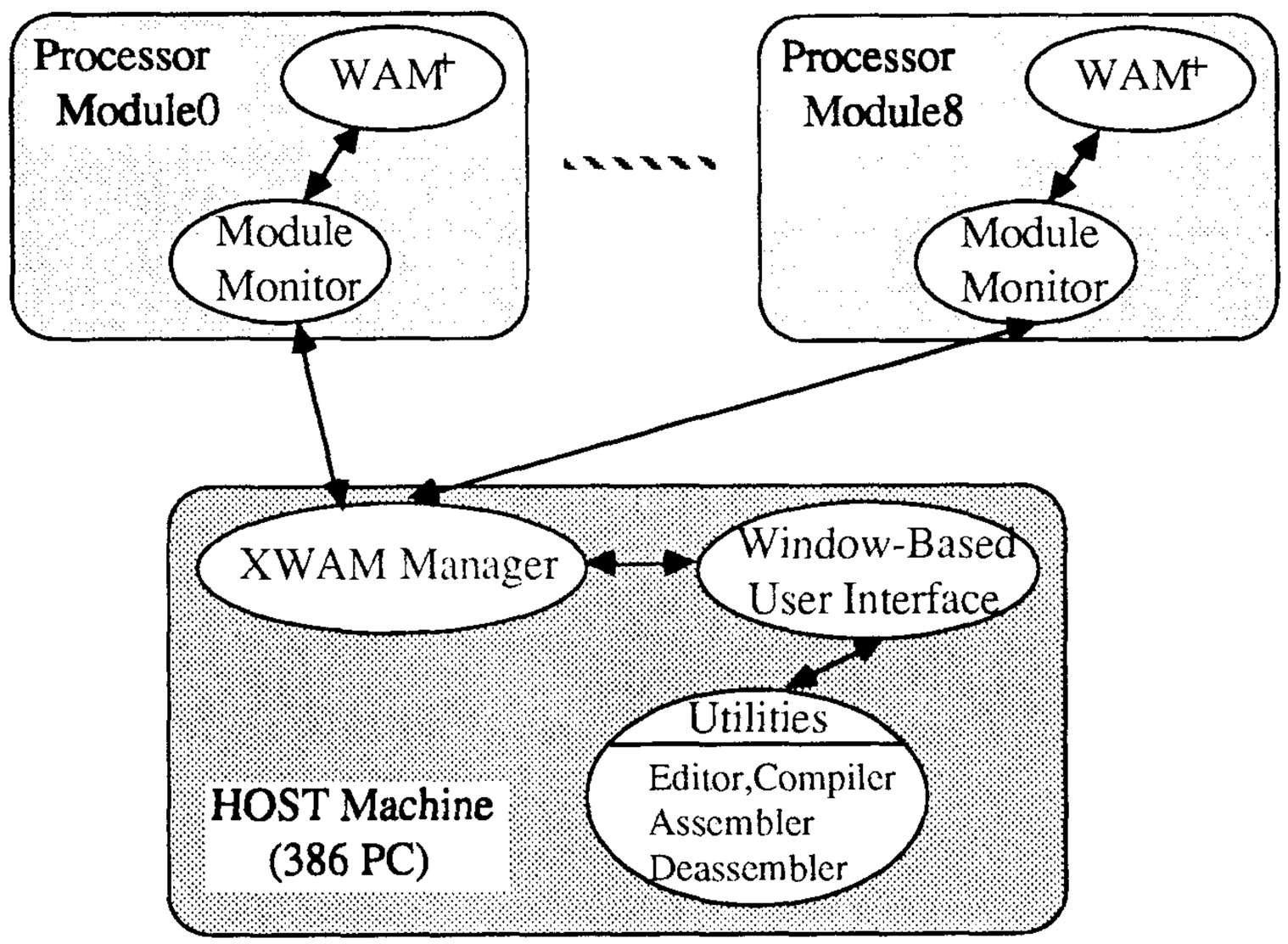



<그림 3.11> 병렬 입출력 모듈

## 3.3 시스템 소프트웨어

### 3.3.1 시스템 소프트웨어의 구성

<그림 3.12>는 X-WAM 시스템 소프트웨어의 전체 구성도를 나타낸 것이다. 각 프로세서 모듈에는 호스트로부터 명령을 받아서 모듈의 수행을 제어하는 모듈 모니터 프로그램이 ROM에 저장되어 있다. 호스트에는 각 모듈을 제어하는 X-WAM 매니저 프로그램과 사용자가 Prolog 프로그램을 개발하기 위한 환경(컴파일러, 어셈블러, 편집기, 그래픽 함수 등)이 있다. 현재 호스트는 높은 해상도 (1024x1180)를 제공하는 컬러 그래픽스 디스플레이와 25 MHz의 386 PC 시스템으로 구성되어 있으며, 사용자 인터페이스는 마이크로 소프트사의 MS-WINDOWS를 사용하여 작성하였다.



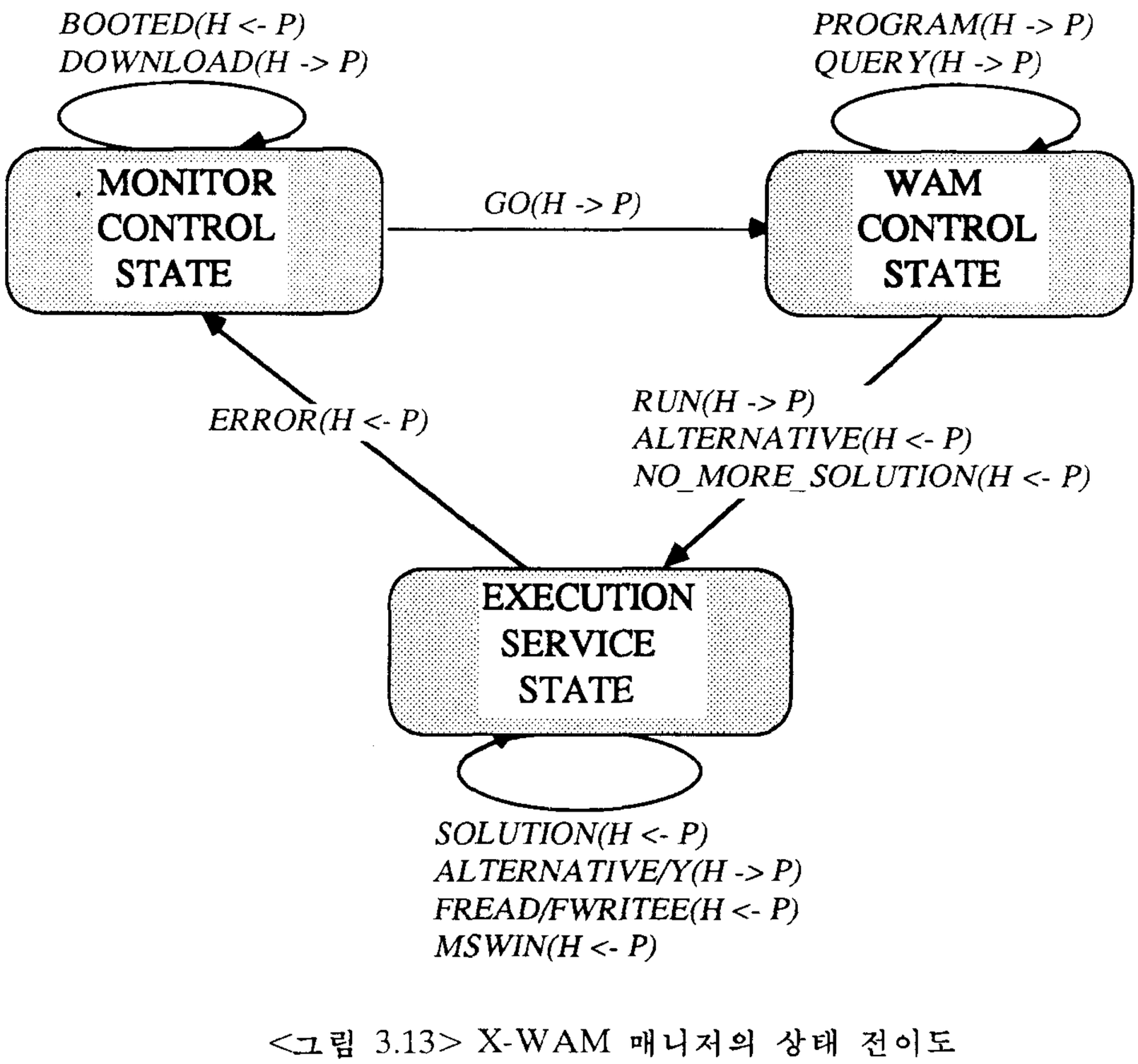

<그림 3.12> X-WAM의 시스템 소프트웨어 구성도

### 3.3.2 X-WAM 매니저와 통신 프로토콜

X-WAM 매니저는 Prolog 프로그램을 여러개의 프로세서 모듈(PE : Processing Element)에서 수행시키기 위하여 각 PE들을 관리하고, 사용자와의 인터페이스를 제공하는

프로그램이다. 기본적으로 X-WAM 매니저에서 제공하는 기능으로는 마우스를 이용한 사용자 인터페이스와 편집기, Prolog 프로그램을 병렬로 수행 시킬 수 있는 기능, 그리고 프로그램의 수행 결과를 그래픽을 이용하여 보여주는 기능 등을 제공한다. 본 절에서는 X-WAM 매니저에서 제공하는 여러가지 기능과 back-end에서 수행되는 PE와의 통신 방법에 대하여 설명하도록 한다.

X-WAM 매니저는 여러개의 PE를 이용하여 Prolog 프로그램을 수행시키기 위하여 여러 상태 사이를 전이하면서 수행한다. <그림 3.13>에 나타낸 것과 같이 X-WAM 매니저의 상태는 크게 MONITOR-CONTROL 상태와 WAM-CONTROL 상태, 그리고 EXECUTION-SERVICE 상태 등의 3개로 나눌 수 있는데, 각 상태에서 수행하는 작업과 이때 발생하는 메시지의 종류, 그리고 각 상태사이의 상태 전이는 다음과 같다. 여러개의 PE를 사용하는 경우에는 같은 메시지를 여러개의 PE로 보내면 된다.




<그림 3.13> X-WAM 매니저의 상태 전이도

- MONITOR CONTROL 상태

1) PE가 boot되어 있나를 검사한다. -- CHECK BOOT 시

2) $WAM^+$ 에뮬레이터를 각 PE에 다운로드시킨다. -- DOWNLOAD X-WAM 시

3) $WAM^+$ 에뮬레이터를 수행 시킨다. -- RUN X-WAM 시

- WAM CONTROL 상태

1) 응용 프로그램을 $WAM^+$ 에뮬레이터의 코드 영역에 적재한다. -- LOAD 시

2) 사용자로 부터 질의어를 받아서 컴파일한 후, 이것을 $WAM^+$ 에뮬레이터안의 코드 영역에 적재한다 -- QUERY 시

3) 각 PE에 적재되어 있는 응용 프로그램의 수행을 시작한다. -- GO APPLICATION 시

- EXECUTION SERVICE 상태

1) PE로 부터 오는 메시지를 받아서 해석한다. -- DURING EXECUTION 시

위의 각 단계에서 필요한 메시지의 종류와 순서는 다음과 같다.

(1) CHECK BOOT 시

각 PE는 전원이 들어 오면 X-WAM 매니저에게 BOOTED 메시지를 보내고, X-WAM 매니저는 각 PE의 BOOTED 메시지를 검사함으로써 현재 사용가능한 PE의 갯수를 알 수 있다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : none

PE --> X-WAM 매니저 : BOOTED

(2) DOWNLOAD X-WAM 시

68000 코드로 변환된 $WAM^+$ 에뮬레이터를 각 PE에게 적재하기 위해서는 먼저 DOWN_LOAD라는 메시지를 각 PE에게 보내고, 다음과 같은 T-RECORD 레코드를 이용하여 $WAM^+$ 에뮬레이터를 각 PE에 전송한다. 각 PE는 T-RECORD를 받을 때마다 check-sum을 계산하여 에러가 생겼는지 계속 검사하고, T-RECORD의 헤드에 TEND가 있으면 그 결과를 X-WAM 매니저에게 ACK 메시지를 이용하여 보낸다. X-WAM

매니저는 $WAM^+$ 에뮬레이터를 모두 보내고 난 다음에는 각 PE의 다운로드 상태를 검사한다.

- T-RECORD 형식

| TDATA(TEND) | LENGTH | ADDR | CODE | CHECK_SUM |
|---|---|---|---|---|

- 메시지의 종류 :

X-WAM 매니저 --> PE : DOWN_LOAD

PE --> X-WAM 매니저 : ACK

(3) RUN X-WAM 시

각 PE에 다운로드된 $WAM^+$ 에뮬레이터를 수행시키기 위해서 X-WAM 매니저는 각 PE에게 GO라는 메시지와 에뮬레이터 시작번지를 보내고 WAM-CONTROL 상태로 전이 한다. 각 PE는 GO 메시지와 시작 번지를 받으면 ACK 메세지를 보내고 시작 번지에 있는 에뮬레이터를 수행시켜서 WAM-CONTROL 상태로 전이된다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : GO

PE --> X-WAM 매니저 : ACK

(4) LOAD 시

X-WAM 매니저는 사용자가 선택한 응용 프로그램의 컴파일된 오브젝트 화일을 읽어서 각 PE에게 보낸다. 보내는 방법은 우선 PROGRAM이라는 메시지를 보내고 난후에 Pno(전체 PE의 갯수), 번치(bunch : 전체 PE중에서의 해당 PE의 일련 번호) 모우드(all solution mode 또는 single solution mode), 오브젝트 코드에 대한 정보(코드의 길이와 코드)를 보낸다. 각 PE는 이런 정보를 받은 다음 응답으로서 ACK 메시지를 X-WAM 매니저에게 보낸다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : PROGRAM

PE --> X-WAM 매니저 : ACK

(5) QUERY 시

X-WAM 매니저는 사용자로부터 고울을 받아 컴파일한 후, 그 코드가 저장되어 있는 goal.obj 화일을 읽어서 각 PE에게 보낸다. 보내는 방법은 우선 QUERY 메시지를 보내고 난 후에 고울 변수에 대한 정보(고울 변수의 갯수와 고울 변수), 함수 테이블에 대한 정보 등을 보낸다. 각 PE는 이런 정보를 받은 다음 응답로서 ACK 메시지를 X-WAM 매니저에게 보낸다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : QUERY

PE --> X-WAM 매니저 : ACK

(6) RUN APPLICATION 시

X-WAM 매니저가 각 PE의 $WAM^+$ 에뮬레이터에 적재된 응용 프로그램을 수행시키기 위해서는 각 PE에게 RUN 메시지를 보내고 PE로부터 응답 메시지를 받은 다음 EXECUTION-SERVICE 상태로 전환한다. 각 PE는 RUN 메시지를 받으면 ACK을 보내고 응용 프로그램의 수행을 시작한 다음 EXECUTION-SERVICE 상태로 된다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : RUN

PE --> X-WAM 매니저 : ACK

(7) DURING EXECUTION 시

X-WAM 매니저는 계속 루프를 돌면서 PE의 서비스 요청을 처리하여 그 결과를 PE에게 보내 준다. X-WAM에서 제공하는 서비스의 종류로는 SOLUTION, 화일 입출력(표준 입출력 포함 : FREAD, FWRITE 등), MSWIN, ERROR 등이 있다. 각 서비스에 대하여 자세히 알아보면 다음과 같다.

A) SOLUTION 서비스

각 PE가 구한 해를 X-WAM 매니저에게 보낼 때는 먼저 SOLUTION 메시지를 보낸 다음 스트링 형태로 해를 보낸다. X-WAM 매니저는 현재의 모우드가 single solution 모우드이면 사용자에게 다음 해를 구할 것인가를 문의한 다음 그 결과에 따라 먼저 ALTERNATIVE 메시지를 보내고 YES/NO(NO : 수행을 끝냄, YES : 다음 해를 구함)를 보낸다. X-WAM 매니저나 PE 모두 YES/NO가 YES인 경우나 all solution 모우드인 경우는 EXECUTION SERVICE 상태를 계속 수행하며, YES/NO가 NO인 경우에는 WAM CONTROL 상태로 전환한다. 에뮬레이터에서 더 이상의 해를 구할 수 없으면 PE는 NO_MORE_SOLUTION 메시지를 X-WAM 매니저에게 보내고

WAM CONTROL 상태로 전환하며, X-WAM 매니저도 NO_MORE_SOLUTION 메시지가 오면 WAM CONTROL 상태로 전환한다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : ALTERNATIVE

PE --> X-WAM 매니저 : SOLUTION, NO_MORE_SOLUTION

B) FREAD/FWRITE 서비스

응용 프로그램의 수행 도중에 에뮬레이터가 이미 연 화일로부터 데이타를 읽거나 쓰고 싶을 때 PE는 X-WAM 매니저에게 FREAD/FWRITE 메시지를 보낸다. X-WAM 매니저는 FREAD 메시지를 받으면 현재 열린 화일로 부터 데이타를 읽거나 쓰고 데이타의 길이와 데이타를 DATA 메시지와 함께 PE에서 수행되고 있는 에뮬레이터에게 보낸다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : DATA

PE --> X-WAM 매니저 : FREAD/FWRITE

C) MS_WIN 서비스

응용 프로그램의 수행 도중에 MS-WINDOWS 상에 어떤 그림을 그리고 싶을 때 PE는 X-WAM 매니저에게 MS_WIN 메시지와 함께 그리고 싶은 그림에 대한 정보가 들어 있는 해를 X-WAM 매니저에게 보낸다. X-WAM 매니저는 MS_WIN 메시지를 받으면 다음에 있는 스트링을 받아서 실제 MS-WINDOWS 상의 동작으로 바꾸어주고, 그 결과를 ACK 메시지를 통하여 PE에서 수행되고 있는 에뮬레이터에게 보낸다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : ACK

PE --> X-WAM 매니저 : MS_WIN

D) ERROR 서비스

응용 프로그램의 수행 도중에 오류가 발생하면 PE는 X-WAM 매니저에게 ERROR 메시지와 함께 오류 메시지를 X-WAM 매니저에게 보내고 MONITOR CONTROL 상태로 전환한다. X-WAM 매니저는 ERROR 메시지를 받으면 그 오류 메시지를 사용자에게 보여주고 MONITOR CONTROL 상태로 전환한다.

- 메시지의 종류 :

X-WAM 매니저 --> PE : non
PE --> X-WAM 매니저 : ERROR

### 3.3.3 X-WAM 모듈 모니터 및 $WAM^+$ 에뮬레이터

모듈 모니터(Module Monitor) 프로그램은 각 모듈의 수행을 제어하는 프로그램으로서, 모듈 내의 400000번지 부터 407FFFF까지의 32K ROM에 저장되어 있다. 주요 기능은 시스템 제어에 관한 기능과, 병렬 및 직렬 입출력에 관한 기능, 시스템 진단(메모리 테스트 등)에 관한 기능, $WAM^+$ 에뮬레이터를 RAM 영역에 다운로드하는 기능 등을 가지고 있다. 모듈 모니터는 호스트의 X-WAM 매니저로부터 RUN 메시지를 받으면 제어를 $WAM^+$ 에뮬레이터에게 넘겨주어 Prolog 프로그램을 수행하도록 한다.

Prolog 프로그램은 X-WAM 컴파일러에 의하여 컴파일되어 프로세서 모듈의 RAM 영역에 있는 Prolog의 순차 추상 머신인 WAM(Warren Abstract Machine)의 변형인 $WAM^+$ 에뮬레이터에 의하여 수행된다. $WAM^+$는 OR 병렬성을 추구하기 위하여 WAM에 몇개의 인스트럭션을 첨가한 것이다. 이 에뮬레이터는 프로세서 모듈의 00008번지부터 3FFFFF까지의 RAM 영역에 저장된다. 이 프로그램은 SUN3/160에서 C 언어를 이용하여 작성한 것으로서, MCC68K 라는 도구를 사용하여 MC68000 기계어로 번역되어 호스트로부터 다운로드 된다. 이 에뮬레이터의 입출력 인터페이스는 ROM에 있는 병렬 및 직렬 입출력 기능을 이용하는데, 수행 도중에 필요한 서비스는 호스트에 있는 매니저 프로그램에게 요구하여 수행한다.

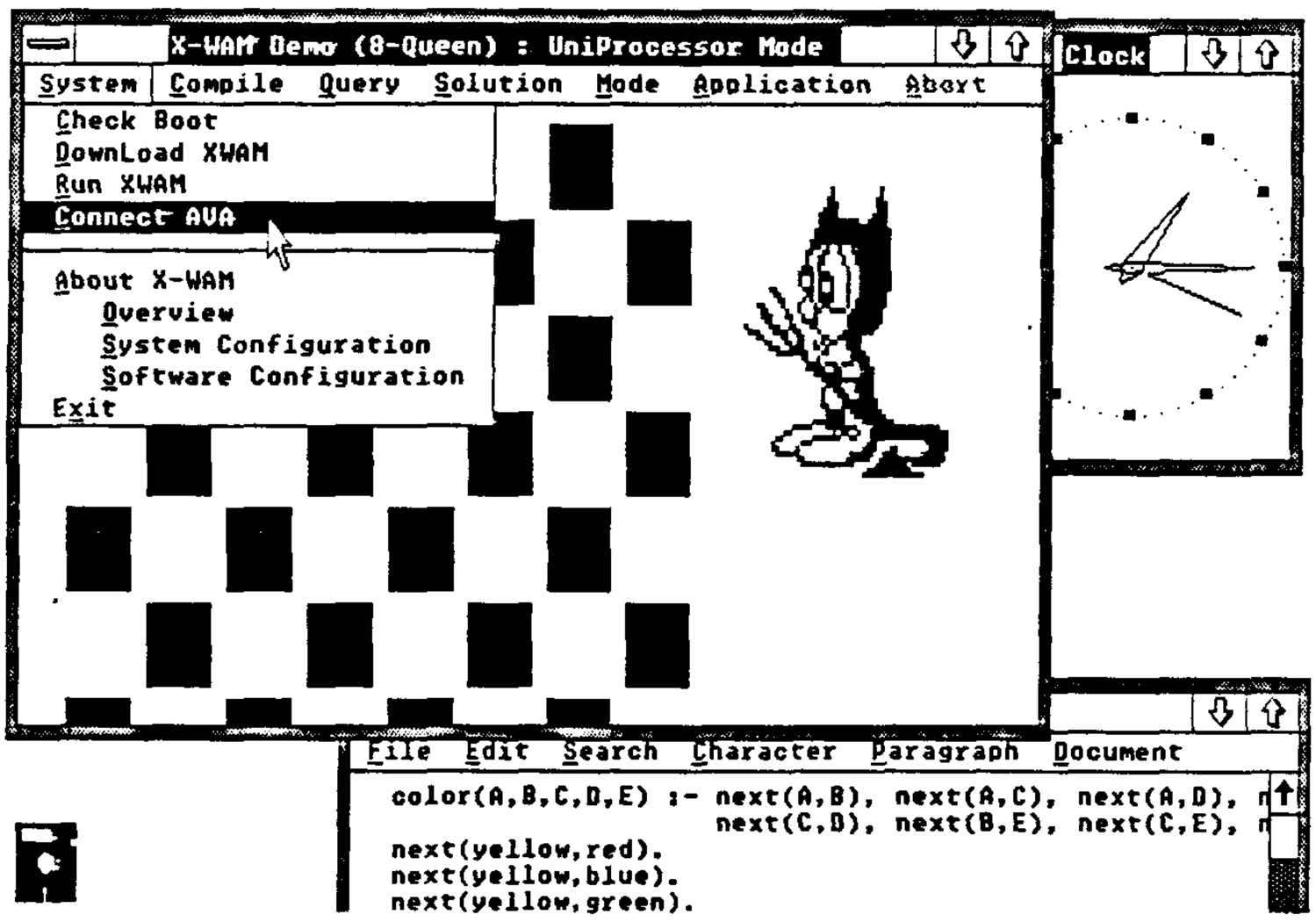
X-WAM 소프트웨어 에뮬레이터는 크게 프로그램 및 질의어의 목적 코드의 입력, 프로그램의 수행 그리고 결과의 출력 루틴으로 구성되어 있다. 프로그램의 수행 루틴은 일련의 인스트럭션의 수행들로 이루어지며 하나의 인스트럭션의 수행은 인스트럭션을 하나씩 페치하고 해석하여 수행하는 것으로 되어 있다. 이 과정은 일반적인 머신이 기계어로 작성된 프로그램을 수행하는 과정과 동일하다. 이에 덧붙여서 수행시 fail이 발생하면 되돌림(backtracking)이 일어나게 되는데 이를 위한 루틴이 있다.

2차년도에 작성한 프로토타입과 다른점은 Prolog의 비논리적인 특징들(데이타베이스 처리 기능, 입출력 기능 등)을 첨가시켰다는 것이다. 실제 Prolog 프로그램들은 이런 특징들을 많이 사용하기 때문에 이 기능은 필수적인 것이다.

## 3.4 소프트웨어 개발 환경

### 3.4.1 윈도우에 기반을 둔 사용자 인터페이스

X-WAM 사용자 인터페이스는 오버랩 윈도우 시스템인 마이크로 소프트 WINDOWS 상에 구현되었으며, 다른 WINDOWS 응용 프로그램과의 공존이 가능하다. 사용자와의 입출력은 마우스 및 키이보드로 제어되며, 처리 결과는 고해상도 컬러 그래픽 디스플레이에 표시된다. <그림 3.14>는 X-WAM의 사용자 인터페이스 최상위 레벨의 한 화면을 나타낸 것이다.




<그림 3.14> X-WAM의 사용자 인터페이스 최상위 레벨 화면

X-WAM 사용자 인터페이스는 부트된 프로세서 모듈중에서 임의의 갯수만을 지정하여 프로그램을 수행시킬 수 있는 기능을 제공한다. 프로그램의 편집 및 컴파일에 관한 기능으로는, 사용자가 요구하면 필요한 유틸리티를 호출하는 형식으로 운영된다. 유틸리티는 프로그램 편집기, 논리언어의 컴파일러, 어셈블러, 디어셈블러, 로우더등이 있다. 편집기는 마우스 제어에 의한 텍스트 편집기로 윈도우 시스템 상에서 함께 운영된다. 프로그램 로더는 컴파일된 Prolog 코드를 각 프로세서의 에뮬레이터 메모리 영역에 적재시키기 위한 것이다. 현재 X-WAM 사용자 인터페이스에서 제공하는 기본적인 메뉴는 다음과 같다.

(1) **SYSTEM** : 시스템에 관련된 메뉴

- *Check Boot* : back-end에서 수행되는 PE의 갯수를 검사하고, 그들을 초기화 시켜준다.
- *DownLoad X-WAM* : 68000 코드로 작성된 $WAM^+$ 에뮬레이터를 각 PE에 적재시켜 준다.
- *Run X-WAM* : 각 PE에 적재된 $WAM^+$ 에뮬레이터를 수행시킨다.
- *About X-WAM* : X-WAM에 대한 전반적인 소개와 시스템 구성, 그리고 소프트웨어 구성에 대한 정보를 보여 준다.
- *Exit* : X-WAM 매니저의 수행을 끝내고 DOS 환경으로 돌아간다.

(2) **COMPILE** : Prolog 프로그램을 컴파일하기 위한 메뉴

- *Compile* : 주어진 Prolog 프로그램을 $WAM^+$ 오브젝트 코드로 컴파일하여 *.obj 화일에 저장한다.
- *Assemble* : 주어진 $WAM^+$ 심볼 코드를 오브젝트 코드로 만든다.
- *Deassemble* : $WAM^+$ 오브젝트 코드를 심볼 코드로 바꾸어 준다.
- *Load* : 컴파일된 Prolog 프로그램(*.obj 형태)을 여러개의 PE에게 보낸다.

(3) **EDITOR** : 사용자가 윈도우 상에서 프로그램을 작성할 수 있도록 편집기를 호출한다.

(4) **QUERY** : 사용자로부터 고울을 받아서 컴파일한 후, 그 코드를 각 PE에게 보낸다.

(5) **Mode** : Prolog 프로그램의 수행을 하나의 PE만을 사용할 것인지, 여러개의 PE를 사용할 것인지를 결정하는 메뉴이다.

- *Uni-Processor* : 하나의 프로세서만을 사용하여 수행
- *Multi-Processor* : 사용자의 선택에 따라 여러개의 프로세서를 이용하여 수행

(6) **Application** : Prolog 프로그램의 수행 결과를 그랙픽을 이용하여 보여주기 위한 메뉴로서, 현재 X-WAM 매니저에서 그래픽을 이용하여 보여줄 수 있는 응용 프로그램은 다음과 같다.

- 8 Queen
- Map Coloring
- Seoul Subway
- Sphinx
- General Application

  이때 Genral Application 메뉴를 선택하면 어떤 프로그램이라도 텍스트 형태로 수행 결과를 볼 수 있다.

### 3.4.2 K-Prolog 컴파일러

K-Prolog 컴파일러는 Prolog 원시 프로그램을 X-WAM 인스트럭션의 바이트 코드(Byte Code)로 구성된 목적 코드로 컴파일한다. 이 컴파일러에 의해 생성된 목적 코드는 소프트웨어 에뮬레이터에 의해 직접 수행가능한 코드이다. 이 절에서는 본 컴파일러의 입력 언어인 Prolog 언어와 머신언어인 X-WAM 인스트럭션에 대하여 설명한다.

#### 1) Prolog 언어

- 구문 구조 *(Syntax)*

대부분의 Prolog 구문 구조는 C-Prolog와 거의 같다. BNF 형태의 구문은 <부록-5>에 나타나 있다. 그러나 본 시스템에서 제공하고 있는 내장 술어(Built-In Predicate, System Predicate)는 C-Prolog에 비해 다소 제한되어 있으며 그들의 구문에 있어서도 약간의 차이가 있다.

- 내장 술어 *(Built-in Predicates)*

<테이블 3.1>은 X-WAM 컴파일러가 제공하고 있는 내장 술어들을 나타낸 것인데, 가장 널리 쓰이는 C-Prolog와의 구문 차이도 함께 나타내었다.

| (C-Prolog) | (K-Prolog) |
|---|---|
| Unify | |
| = | equal(term1,term2) |
| is | is |
| == | eq(term1,term2) |
| =\= | ne(term1,term2) |
| Arithmetic | |
| =:= | =:= |
| =\= | =\= |
| < | < |
| > | > |
| =< | >= |
| >= | >= |
| + | + |
| - | - |
| * | * |
| / | / |
| Input/Output | |
| nl | nl |
| put | put |
| get0 | get0 |
| get | get |
| tab | tab |
| read | read |
| write | write |
| see | see |
| seeing | seeing |
| seen | seen |
| tell | tell |
| telling | telling |
| told | told |

| (C-Prolog) | (K-Prolog) |
|---|---|
| Internal Database | |
| recorda | recorda |
| recordz | recordz |
| recorded | recorded |
| erase | erase |
| Higher-Order | |
| call | call |
| =.. | univ(term,list) |
| functor | functor |
| arg | arg |
| name | name |
| setof | setof |
| findall | findall |
| Test | |
| integer | integer |
| atom | atom |
| var | var |
| nonvar | nonvar |
| atomic | atomic |
| true | true |
| fail | fail |

<테이블 3.1> K-Prolog의 내장 술어

2) X-WAM 인스트럭션 집합

X-WAM 인스트럭션은 Warren Abstract Machine(WAM)의 인스트럭션을 약간 수정하여 설계되었다. 가장 다른 점은 OR 병렬을 구현하기 위한 인스트럭션으로서 OR 병렬성을 위한 *entry-point*와 AND 병렬성을 위한 여러 인스트럭션이 있다. X-WAM의 인스트럭션 집합은 <부록-6>에 나타나 있다. X-WAM 인스트럭션이 WAM 인스트럭션과 비교하여 특징적인 것을 보면 아래와 같다.

*X-WAM* 인스트럭션의 특징

- AND/OR 병렬성을 구현하기 위한 인스트럭션의 추가
- Cut을 지원한다
- Top-Down 방식의 Structure Unification을 지원한다
- One-Level Indexing Mechanism을 사용한다

질의어의 컴파일은 프로그램의 컴파일과 분리되어 행하여진다. 이를 위해서 프로그램의 컴파일시에 목적 코드상에서의 각 프로시듀어의 주소와 프로그램상에 나타나는 functor 및 atom에 관한 정보를 특별한 화일에 저장해 두고 있다. 이 정보는 질의어의 컴파일시에 질의어의 머신 코드와 프로그램의 머신 코드를 링크시킬 때 사용한다.

### 3.4.3 X-WAM 어셈블러

X-WAM 어셈블러는 X-WAM의 기호 인스트럭션으로 구성된 프로그램을 미리 정의된 바이트 코드로 어셈블한다. 사용된 바이트 코드는 X-WAM 인스트럭션과 1:1 대응하는 것으로서 바이트 코드 에뮬레이터에 의해 수행된다. 정의된 바이트 코드는 <부록-7>에 나타나 있다. 다음은 append 프로그램을 기호 코드로 나타낸 것이다.

```
        halt
L2:     try_me_else_L L14
L4:
        get_nil_A X1
        get_variable_X X4, X2
        get_value_X X4, X3
        proceed
L14:    trust_me_else_fail
L16:allocate
        get_list_A X1
        unify_variable_X X4
        unify_variable_Y Y3
        get_variable_Y Y2, X2
        get_list X3
        unify_value_X X4
        unify_variable_Y Y1
        put_value_Y Y3, X1
        put_value_Y Y2, X2
        put_value_Y Y1, X3
```

```
        deallocate
        execute_PN append/3, 3
append/3: switch_on_term_L L2, L4, L16, fail
```

위 기호 코드를 어셈블링한 결과는 아래와 같다.

```
상대 주소    바이트 코드

01          7
02          44  13
04          13  1
06          8   4   2
09          10  4   3
12          3
13          46  0
15          4
16          16  1
18          28  4
20          29  3
22          9   2   2
25          16  3
27          30  4
29          29  1
31          20  3   1
34          20  2   2
37          20  1   3
40          5
41          2   44  3
44          47  2   4   15  0
```

### 3.3.4 X-WAM 디어셈블러

X-WAM 디어셈블러는 컴파일러나 어셈블러에 의해 생성된 바이트 코드를 X-WAM 기호 인스트럭션으로 변환시키는 유틸러티이다. 이 프로그램은 Prolog 프로그램의 X-WAM 인스트럭션 형태를 기호의 형태로 볼 수 있어 X-WAM을 이해하는데 도움을 주며 Prolog 프로그램 수행을 어셈블리 언어수준에서 추적해 볼 수 있게 해준다.

# 제 4 장 Prolog 전용 프로세서의 개발

## 4.1 개 요

X-WAM의 각 프로세서 모듈은 Warren Abstract Machine(WAM) [War83]을 구현한 순차 추론 머신이다. WAM을 구현하는 방법은 크게 두 가지로 분류할 수 있다 [War83, Tick87a]. 첫 번째 방법은 범용 프로세서를 사용하여 구현하는 방법으로서, Prolog 프로그램을 WAM 인스트럭션으로 컴파일한 후 다시 그 프로세서의 기계어로 컴파일하여 수행시키는 방법 [Carl86], Prolog 프로그램을 바이트코드 형태의 WAM 인스트럭션으로 컴파일하여 소프트웨어 에뮬레이션하는 방법 [Tick87b, Ryu87], 또는 WAM 코드로 컴파일된 코드를 범용 컴퓨터에서 마이크로 코드에 의해 해석하는 방법 [Gee87]등이 이에 속한다.

현재 X-WAM의 각 프로세서 모듈은 범용 single board 컴퓨터에서 WAM 인스트럭션을 소프트웨어 에뮬레이션하는 방법을 사용하여 WAM을 구현한 것이다. 이러한 방법은 비용이 적게 들고 이미 개발되어 있는 빠른 속도의 범용 컴퓨터를 이용할 수 있는 장점이 있으나 절차적 언어의 특성에 알맞게 설계된 범용 컴퓨터와 Prolog 언어 사이에는 semantic gap이 있어서 수행 속도의 개선에는 한계가 있다 [Tick87a].

WAM을 구현하는 두 번째 방법은 WAM 구조에 적합한 전용 프로세서를 설계하여 WAM 인스트럭션을 하드웨어 또는 마이크로 프로그램에 의해 수행하는 방법이다. 이 방법은 추가적인 하드웨어가 필요하지만 첫 번째 방법에 비해 궁극적으로 높은 성능을 얻을 수 있으며, Prolog가 가지고 있는 언어상의 특징을 하드웨어로 지원 받을 수 있는 장점이 있어서, 병렬 추론 머신을 개발하기 위한 세계 각국의 연구에서 프로세서 모듈은 전용 프로세서를 사용하고 있다 [Dobr85, Ichi87, Baro88, Warr88].

WAM에 기반을 둔 Prolog 머신에는 UC Berkeley의 PLM [Dobr84, Dobr85]과 일본에서 개발된 CHI [Nakaz86], PSI-II [Nakas87] 등이 있다. 이상의 머신들은 Prolog 전용 머신으로서 높은 성능을 보였지만 아직 개선할 점이 많다. 예를 들면 Prolog를 수행할 때 가장 빈번히 수행되고 또 수행시간이 많이 걸리는 연산이 단일화 연산임이 여러 시뮬레이션 결과로 밝혀졌으며 [Woo85, Dobr85, Tick87b], 따라서 단일화와 관련된

연산들을 보다 빠르게 수행시킬 수 있는 머신이 요구되는 데 위에서 열거한 머신들은 단일화 연산을 수행하는데 상대적으로 많은 시간이 걸리고 있다.

단일화 연산을 하드웨어 수준에서 지원함으로써 성능을 높이고자 하는 최근의 연구로는 CAM (Content Addressable Memory)을 사용한 특수한 메모리 구조를 제안하는 방법과 multiple functional unit을 사용하여 단일화 및 booking 연산의 fine grain 병렬성을 이용하는 방법으로 나눌 수 있다. 본 연구는 후자와 유사한 접근 방식을 취하고 있다.

단일화는 첫째 데이타 term의 type-checking과 dereferecing, 둘째 dereferencing된 데이타 term의 type-checking, 세째 이러한 term들의 type에 따라 값을 비교하거나 변수를 바인딩하고, 네째 변수의 바인딩이 일어나게 되면 바인딩된 변수들 중에서 trail stack에 저장할 필요가 있는 변수를 trailing하는 과정으로 이루어져 있다. 본 연구에서는 이러한 단일화의 각 과정을 빠르게 수행시킬 수 있는 functional unit을 설계하여 전용 프로세서의 성능을 높이고 궁극적으로는 현재 X-WAM의 프로세서 모듈을 대체함으로써 전체적인 성능을 향상시키는 것을 목적으로 한다.

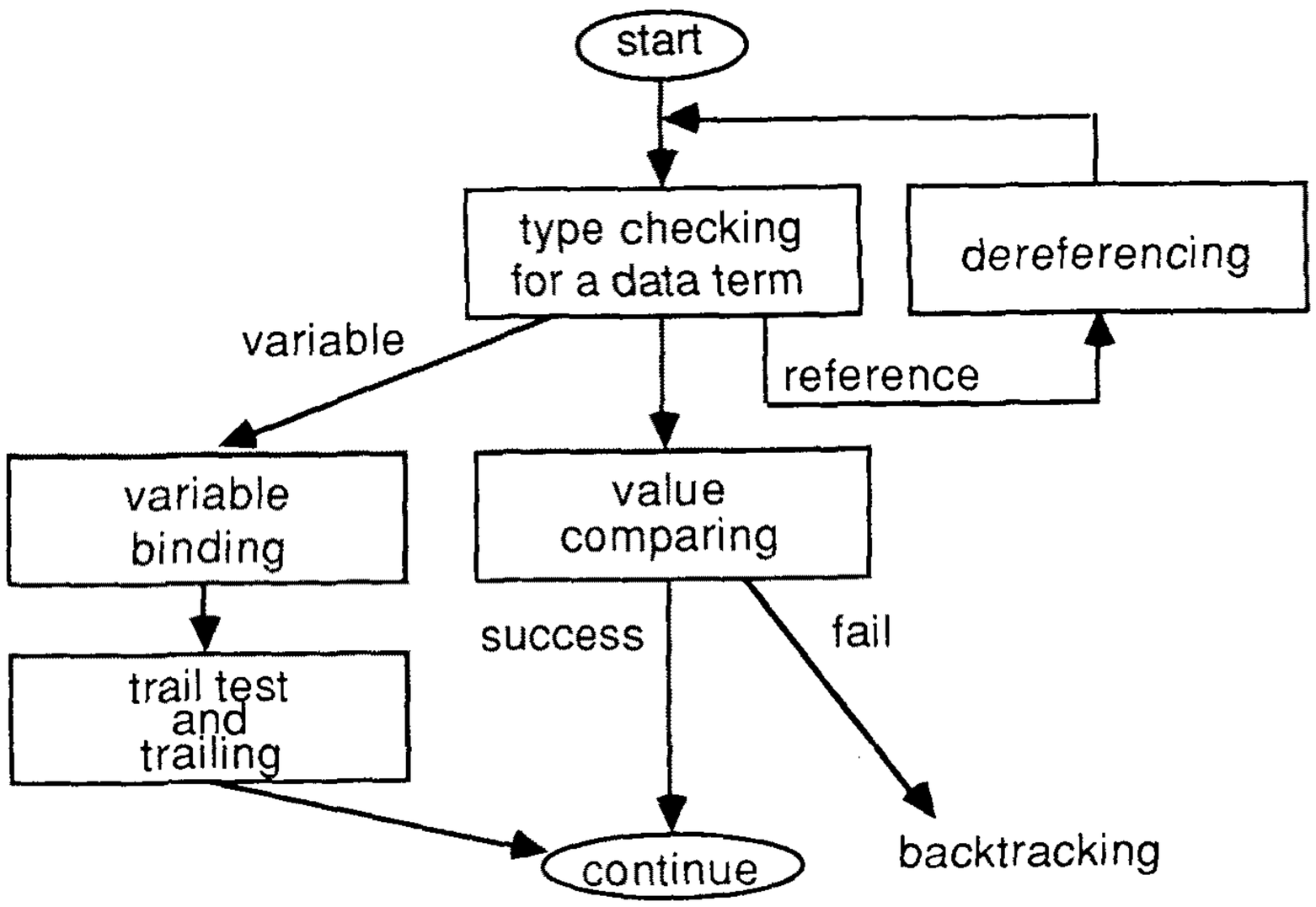
본 장에서는 단일화의 수행 속도를 높이기 위하여, 단일화와 관련된 WAM 인스트럭션들이 수행하는 type-checking 연산을 하드웨어로 지원하며 general unify 연산을 위한 하드웨어 및 변수의 바인딩이 일어날 때 자동적인 trailing를 위한 하드웨어를 첨가한 Prolog 프로세서 APP (A Prolog Processor)를 제안한다. APP는 호스트 프로세서에 의해 WAM 인스트럭션 집합으로 컴파일 된 Prolog 프로그램을 마이크로 프로그램에 의해 수행하는 back-end 프로세서의 형태로 설계되었으며, 메모리와 Processing Unit, Control Unit, 그리고 호스트와의 통신을 위한 하드웨어로 구성되어 있다. Processing Unit은 언급한 하드웨어 외에 데이타와 주소의 동시 전송을 위한 multple bus를 사용하여 데이타 경로상의 데이타 전송의 병렬성을 최대한 높임으로써, 하나의 WAM 인스트럭션을 처리하는 사이클 타임을 줄이는 데에 역점을 두었다.

본 장의 구성은 다음과 같다. 4.2절에서는 APP 설계시의 고려 사항에 대해 기술하고 4.3절에서는 APP의 구조와 하드웨어 구성에 대하여 설명한다. 4.4절에서는 시뮬레이션에 의한 결과를 분석한다.

## 4.2 APP 설계시의 고려 사항

### 4.2.1 단일화의 수행 속도 개선

단일화는 Prolog 프로그램을 수행할 때 가장 빈번히 수행되고 또 수행 시간이 많이 걸리는 연산이다. 따라서 단일화와 관련된 WAM 인스트럭션을 빠르게 수행할 수 있으면 전체적인 성능을 높일 수 있다. <그림 4.1>에 단일화의 수행 과정을 나타내었다.




<그림 4.1> 단일화의 수행 과정

본 연구에서는 단일화와 관련된 WAM 인스트럭션의 수행 속도를 개선하는 방법으로써 <그림 4.1>의 각 과정 중 type-checking과 value-comparing을 지원하는 하드웨어와 trail test 및 trailing을 수행하는 하드웨어를 제안하였다.

#### 4.2.2 다양한 데이타 경로의 제공

WAM에서 사용되는 레지스터들은 메모리의 어떤 장소를 지정하는 "주소"이면서 경우에 따라서는 "데이타"로 사용되기도 한다. 또 이러한 레지스터의 내용들이 전송되는 데이타 경로는 매우 복잡하다. 레지스터들 간의 전송과 메모리와의 데이타로서의 전송이 불규칙하게 발생하며 동시에 이 레지스터 값이 주소로 사용되기도 한다.

APP는 여러 functional unit을 사용하므로 한 마이크로 인스트럭션에서 여러 연산을 수행하기 위해서는 충분한 데이타 경로가 있어야 한다. APP는 데이타 전송의 병렬성을 충분히 지원하는 다양한 데이타 경로를 제공함으로써 데이타 의존성이 있는 연산이나 연속적인 메모리 참조 외의 모든 데이타 전송은 동시에 수행될 수 있도록 하였다.

#### 4.2.3 메모리 구성

Prolog의 수행시에 요구되는 충분한 메모리 대역폭을 제공하기 위해서 APP에서 사용하는 메모리 영역은 Code 영역, Environment stack, Choice point stack, Heap, Trail로 구분하여 동시에 접근될 수 있도록 하였다.

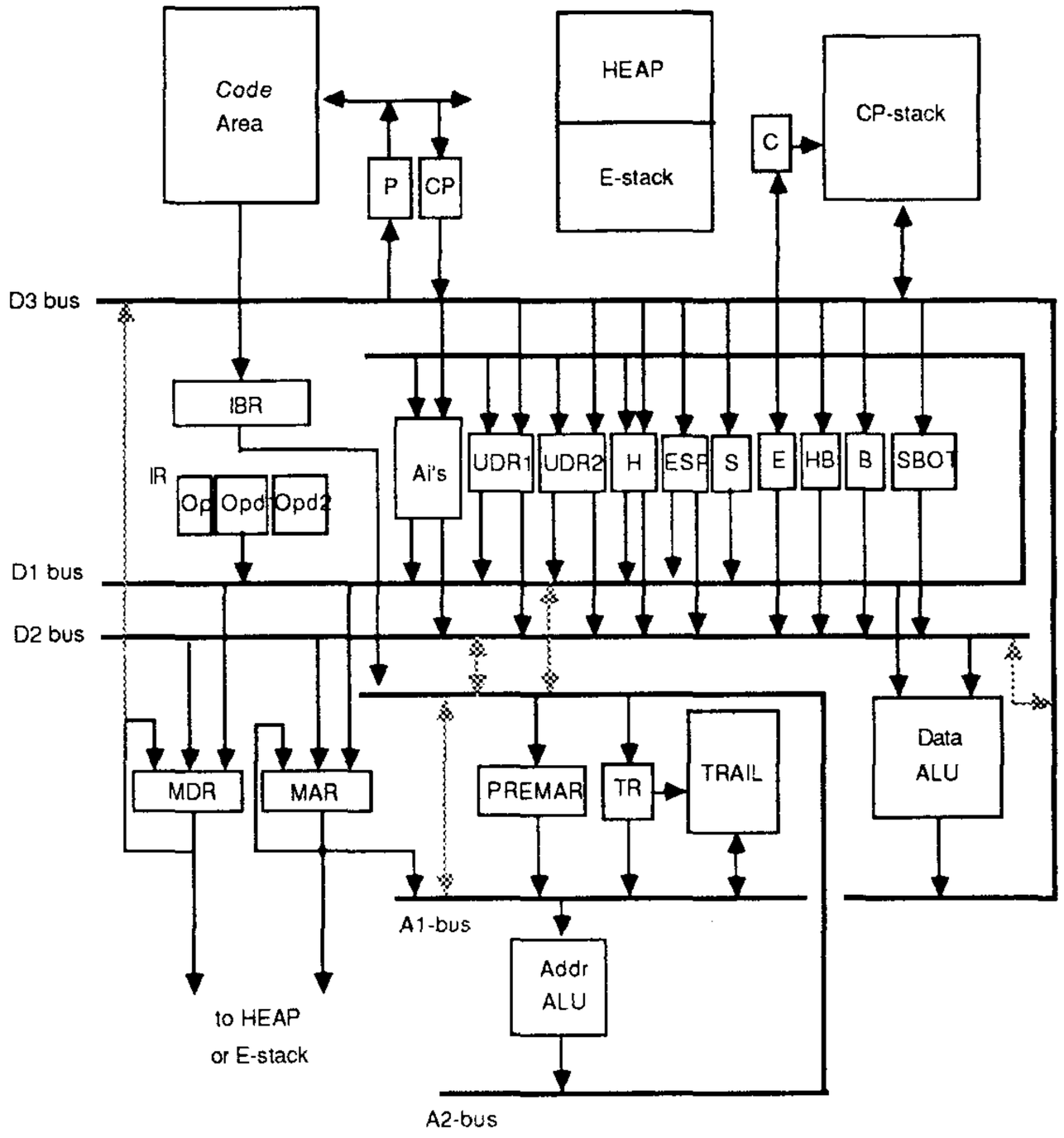
## 4.3 APP의 하드웨어 구성

### 4.3.1 APP의 하드웨어 구성

#### 4.3.1.1 Processing Unit

<그림 4.2>에 Processing Unit의 데이타 경로를 나타내었다. Data ALU는 D1 버스와 D2 버스의 value field에 대해 산술 연산 및 논리 연산을 수행하며 그 결과는 D3 버스를 통해 출력된다. APP에서는 단일화 연산의 value 비교 및 trail 검사는 전용 comparator가 수행하기 때문에 Data ALU는 프로그램에서 요구하는 연산만을 수행한다. Addr ALU는 environment frame의 permanent 변수에 대한 주소 계산을 수행한다. Addr ALU는 Data ALU와 동시에 이러한 연산을 수행할 수 있다. IBR은 인스트럭션을 prefetch하기 위한 레지스터이며 이 중 Operand1 field는 Addr ALU의 입력이 된다. 인수 레지스터는 IR의 Operand1과 Operand2에 의해 선택된다.

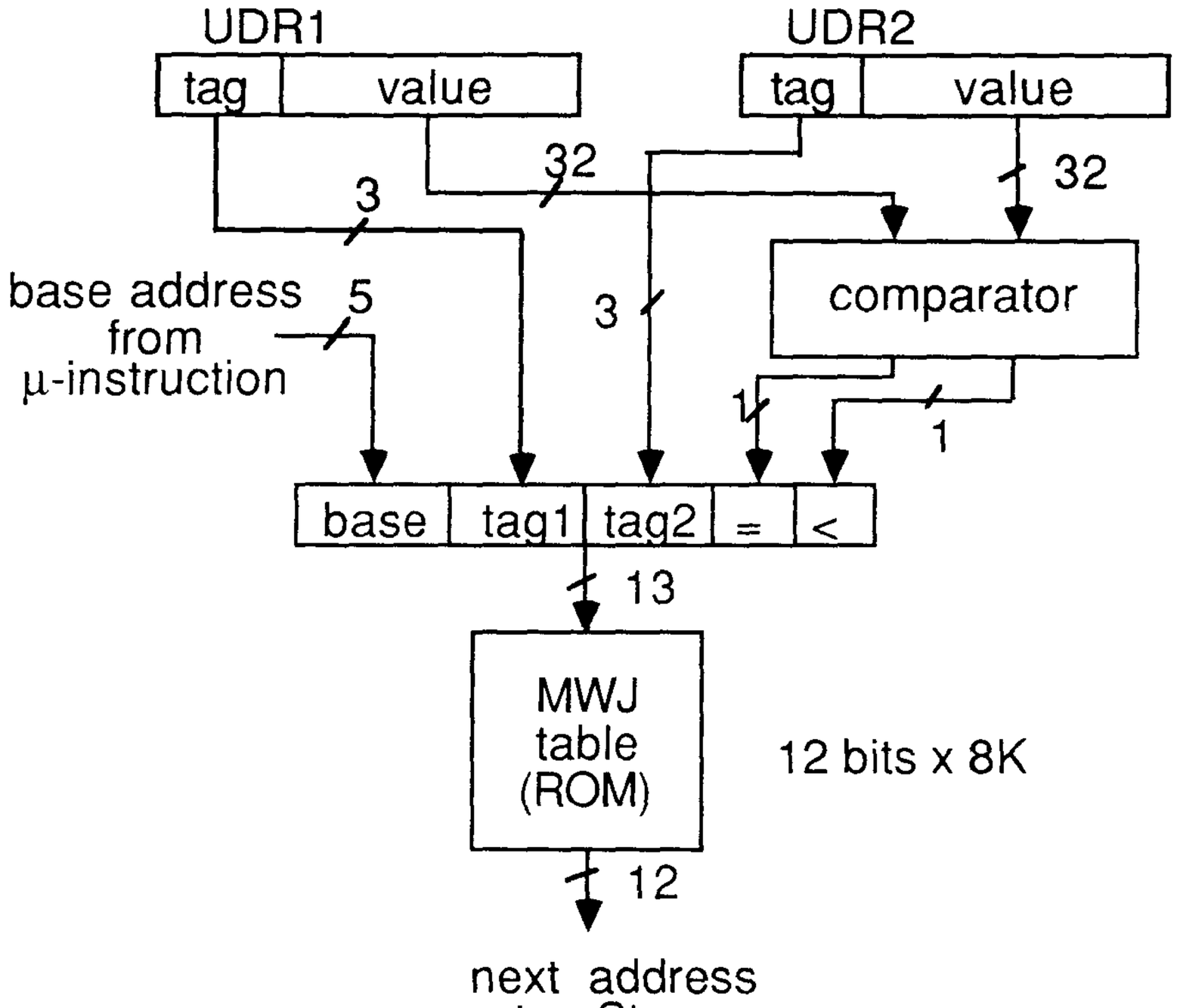
APP는 사용되는 레지스터들의 출력이 여러 하드웨어에 연관되어 사용되므로 대부분의 레지스터는 레지스터 화일로서가 아닌 독립적인 상태로 구현된다. UDR1과 UDR2는 다음 절에서 차례로 설명할 단일화를 위한 하드웨어들의 입력으로 사용되는 데이타를 저장한다.




<그림 4.2> Processing Unit의 데이타 경로

### 4.3.1.2 단일화를 위한 하드웨어

APP는 단일화의 대상이 되는 데이타 term의 tag와 value에 따라 다음에 수행할 마이크로 루틴으로 multi-way jump (MJW)를 하는 데 이의 하드웨어 구성은 <그림 4.3>과 같다.

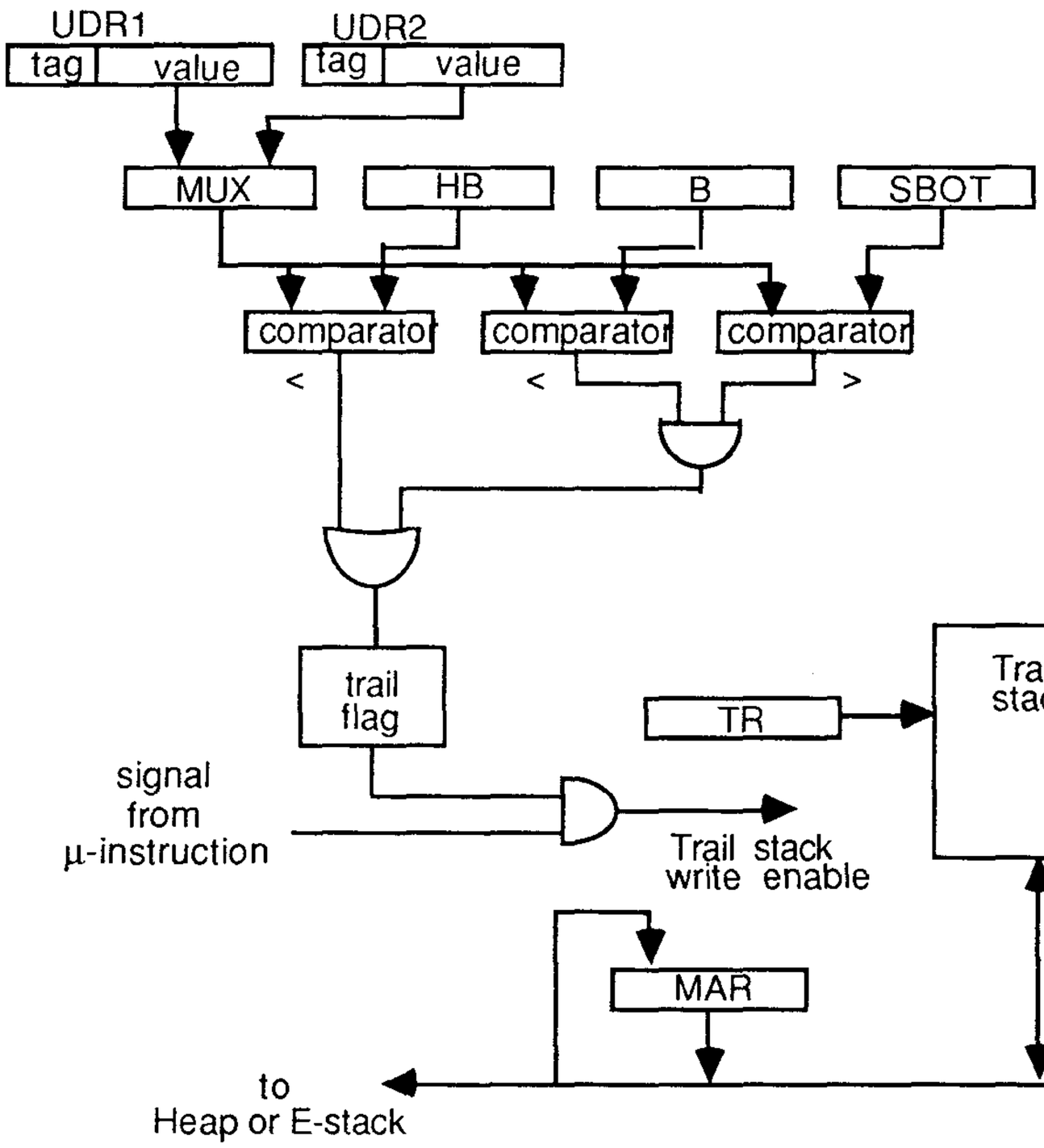



<그림 4.3> 단일화를 위한 하드웨어

Unify 레지스터 UDR1과 UDR2에 저장되어 있는 데이타 term 중에서 tag1과 tag2 두 term의 value field의 상대적인 크기를 나타내는 2 비트, 그리고 마이크로 인스트럭션에서 제공하는 base address 5 비트가 합쳐져 MWJ 테이블의 입력이 된다. 각 인스트럭션에 대하여 MWJ 테이블은 제어 메모리를 가리키는 128개의 12 비트 주소를 제공하며 8K 단어의 ROM에 저장된다.

#### 4.3.1.3 Auto Trailing을 위한 하드웨어

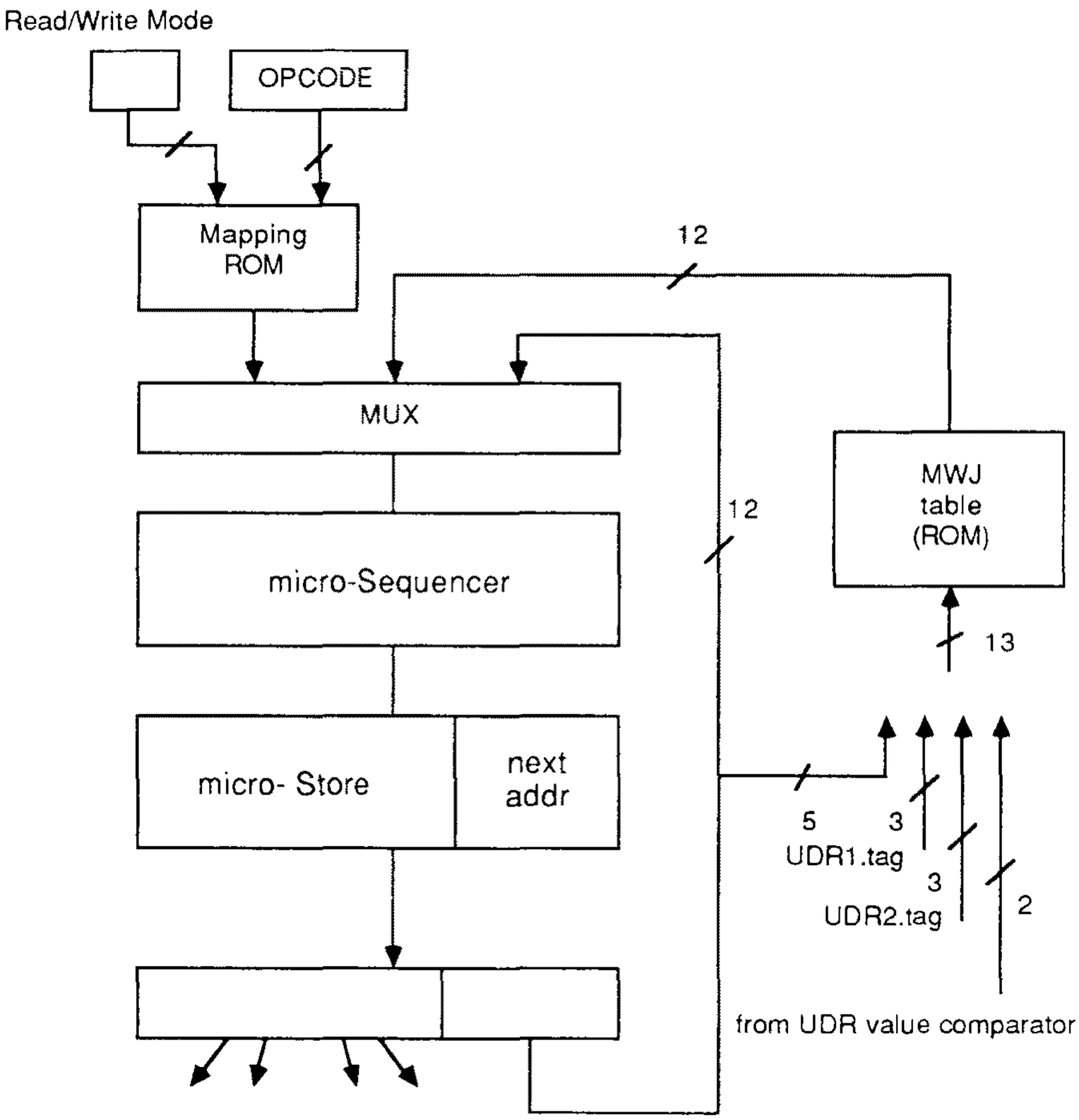
변수의 바인딩이 일어날 때 이 변수의 주소에 대한 trail test를 해야하는 데 APP는 이를 위한 하드웨어를 부착하여 trail test에 따른 Processing Unit의 부담을 제거하였다. Auto Trailing을 위한 하드웨어를 <그림 4.4>에 나타내었다.




<그림 4.4> Auto Trailing을 위한 하드웨어

### 4.3.2 Control Unit

Control Unit Processing Unit의 각 데이타 경로에 signal을 보내어 데이타 전송을 제어하고 ALU의 연산과 마이크로 인스트럭션의 sequence를 결정한다. <그림 4.5>에 Control Unit의 구성을 나타내었다.




<그림 4.5> Control Unit의 구성

## 4.4 시뮬레이션 및 결과 분석

### 4.4.1 성능 평가 및 기존의 Prolog 머신들과의 비교

<표 4.1>에 메모리 참조에 걸리는 시간을 고려하여 기존의 머신들과 성능을 비교하였다. APP의 머신 사이클은 100nsec (10MHz)를 가정하였으며 측정된 결과는 KLIPS 단위로 나타내었다. KLIPS는 전체 수행 시간에 대한 Logical Inference의 횟수에 의해 산출된다. <표 4.1>의 괄호 안의 숫자는 메모리 참조에 걸리는 머신 사이클 수를 나타낸다.

| | APP (1) | PLM (1) | APP (3) | PLM (3) | CHI (1) | PSI-II (1) |
|---|---|---|---|---|---|---|
| append5 | 882 | 426 | 361 | 300 | 250 | - |
| append30 | 842 | - | 338 | - | - | 667 |
| nrev30 | 796 | 323 | 319 | 169 | - | 540 |
| qsort50 | 428 | 213 | 264 | 125 | - | 263 |

<표 4.1> APP의 성능 평가 (단위 KLIPS)

APP는 여러가지 하드웨어를 사용하여 ALU가 수행할 연산을 대신하였고 다양한 데이타 경로를 사용하여 데이타 전송의 병렬성을 최대한 높임으로써 한 인스트럭션 (매크로 인스트럭션)에 대한 마이크로 인트럭션의 수를 줄였기 때문에 인스트럭션 간의 데이타 의존성은 주로 메모리 참조에 나타나며 따라서 전체적인 성능은 메모리 참조

시간에 좌우된다.

메모리 참조 시간을 세 사이클로 가정하였을 때 메모리 참조 시간 동안 ALU는 다른 연산을 수행할 수 있으므로 APP의 하드웨어는 상대적으로 큰 효과를 거두지 못한다. <표 4.1>에서 알 수 있듯이 메모리 참조가 한 사이클만에 수행될 경우 APP는 기존의 머신에 비해 높은 성능을 나타내며 따라서 cache 등의 빠른 메모리를 사용하면 높은 성능을 얻을 수 있다.

| 인스트럭션 | 수행 빈도 (%) | 평균 수행 시간 PLM (cycles) | 평균 수행 시간 APP (cycles) |
|---|---|---|---|
| get_list | | | |
| (bound) | 7.27 | 8 | 4 |
| (unbound) | | 12 | 9 |
| get_constant | | | |
| (bound) | 1.83 | 11 | 4 |
| (unbound) | | 11 | 7 |
| get_nil | | | |
| (bound) | 1.29 | 11 | 4 |
| (unbound) | | 11 | 7 |
| get_structure | | | |
| (bound) | 4.11 | 11 | 4 |
| (unbound) | | 13 | 7 |
| get_value | | | |
| X | 1.44 | 12 | 7 |
| Y | | 14 | 10 |
| unify_constant | | | |
| (read_mode) | 3.33 | 14 | 10 |
| (write_mode) | | 2 | 3 |
| unify_value | | | |
| (read_mode) | | | |
| X | | 23 | 15 |
| Y | 4.96 | 26 | 19 |
| (write_mode) | | | |
| X | | 3 | 3 |
| Y | | 6 | 6 |

<표 4.2> 개선된 APP 인스트럭션의 평균 수행 시간

### 4.4.2 단일화의 수행 속도 개선

APP는 본 논문에서 제안한 단일화를 위한 하드웨어를 첨가함으로써 단일화와 관련된 인스트럭션 중 dereference 연산을 수행하는 인스트럭션의 수행 속도를 크게 개선하였다. <표 4.2>에 개선된 APP 인스트럭션의 수행 시간을 PLM의 경우와 비교하여 나타내었다. 이러한 하드웨어들은 궁극적으로는 한 인스트럭션을 처리하기 위한 마이크로 인스트럭션의 갯수를 줄이는 데에 그 목적이 있으며 이에 따라 마이크로 인스트럭션의 수가 줄어 들면, 이 줄어든 마이크로 인스트럭션에서 이 매크로 인스트럭션의 연산들을 모두 수행해야 하므로 데이타 경로상의 데이타 전송의 병렬성이 수반되어야 한다.

APP는 여러개의 버스를 사용하여 다양한 데이타 경로를 제공하고 각 레지스터들은 병렬성을 만족시키기 위하여 필요한 버스에 연결되어 있으며 마이크로 프로그램은 이들간의 데이타 전송을 적절히 제어함으로써 순차적으로 수행되는 마이크로 인스트럭션간에 데이타 의존성이 없는 경우에는 모든 데이타 전송은 동시에 수행될 수 있게 하였다.

## 4.5 결론 및 연구 방향

본 논문에서는 단일화의 수행 속도를 높이기 위하여 여러 가지 하드웨어를 첨가한 프롤로그 프로세서를 설계하고 시뮬레이션을 통하여 그 성능을 분석하였다. APP는 단일화와 관련된 인스트럭션들이 수행하는 type checking과 value comparing 연산을 지원하는 하드웨어 및 변수의 바인딩이 일어날 때 auto trailing를 위한 하드웨어를 갖추고 또 이러한 단일화를 위한 하드웨어들의 효과를 높이기 위하여 다양한 데이타 경로를 제공하여 단일화 연산의 수행 속도를 개선함으로써 전체적인 수행 속도를 향상시켰다.

APP는 Prolog 전용 프로세서로서 현재 X-WAM 프로토타입 시스템의 기본 프로세서를 대체함으로써 X-WAM의 성능을 크게 향상시킬 수 있다.

# 제 5 장 함수 논리 언어를 위한 추상 머신의 설계

본 장에서는 논리 언어의 기능을 확장하기 위하여 논리 언어에 함수 언어의 기능을 첨가 시킨 함수 논리 언어의 구현에 대하여 설명한다.

## 5.1 개 요

논리 언어와 함수 언어는 대표적인 선언적(declarative) 언어들로서, 각 언어는 다른 언어가 가지고 있지 않는 특징을 가지고 있다. 즉, 논리 언어는 선언적 의미와 관계(relation)를 위주로 한 언어이기 때문에 지식을 바탕으로 하는 추론 분야에 적합한 반면에, 함수 언어는 여러가지 강력한 프로그래밍 개념(예를 들면, high order function, streams, 타입 등)을 제공하며, 또한 알고리즘을 구현하거나 자료 구조를 처리하는데 논리 언어보다는 적합하다 [Bell86]. 따라서 위에서 언급한 두 언어의 장점을 모두 포함하고 있는 새로운 함수 논리 언어(functional logic language)를 개발하려는 연구들이 많이 진행되고 있다[Barb86, Bosc86, Darl86, Gogu86, Levi87, Redd86, Subr86].

함수 논리 언어에 대한 연구는 크게 두 가지로 나눌 수 있는데, 첫번째 부류는 함수 언어에서 사용되는 패턴 매칭(pattern matching)을 단일화(unification)로 확장하는 부류(*functional plus logic*)이며, 두번째 부류는 논리 언어의 단일화에 같음(equality) 관계를 첨가하여 확장한 같음-단일화(Equality-Unification)를 사용하는 부류(*logic plus functional*)이다 [Bosc85]. 함수 논리 언어에 대한 자세한 내용은 [Bell86]에 자세히 나와있다.

위에서 언급한 두가지 부류의 함수 논리 언어는, 다른 언어가 가지고 있지 못하는 여러가지 특징을 가지고 있지만, 아직까지 수행 알고리즘만이 개발되어 있거나, 메타 인터프리터로만 구현되어 있어 효율성이 떨어지기 때문에 실제 응용 분야에서 사용하기는 많은 문제점을 가지고 있다. 첫번째 부류의 언어 중에서, Reddy는 [Redd86]에서 SECD 머신에 기초를 둔 추상 narrowing 머신을 개발하여 어느 정도의 효율성을 제공하였다. 하지만 두번째 부류의 언어에서는 Eqlog[Gogu86], K-LEAF[Bell87, Levi87])와 같이 같음-단일화 알고리즘만이 제안되어 있거나, Funlog[Subr86]나 Aflog[Shin87,Shin88]와 같이 메타 인터프리터로만 구현되어 있어서 실제 응용 분야에 사용하기 어렵다. 이런 문제는 언어의 수행에 필요한 기본 기능을 머신 레벨에서 충분히 제공 하지 못하기 때문에 발생하며, 이 문제를 해결하기 위해서는 함수 논리 언어 수행에 필요한 기본 기능을 머신 레벨에서 제공하여 주어야 한다. 즉, 함수 논리 언어의 수행에 필요한 기본 기능인 resolution과 리덕션을 머신 레벨에서 제공하여 주는 추상 머신이 필요하나 아직 이에 대한 연구는 미흡한 실정이다.

본 연구에서는 이러한 문제점을 해결하기 위하여, 두번째 부류에 속하는 함수 논리 언어의 하나인 Aflog[Shin87, Shin88] 언어에 대한 추상 머신 F-WAM(Functional - Warren

Abstract Machine)의 구조와 인스트럭션 집합을 설계하고, 시뮬레이션을 통하여 성능을 평가하였다. 본 연구에서 F-WAM 설계에 사용한 접근 방법은 다음과 같다. 첫째로는 논리 언어의 효율적인 추상 머신인 WAM(Warren Abstract Machine)에 리덕션 기능을 첨가시켜서 F-WAM을 설계하였다는 것과, 함수 응용을 계산하는 방법으로 "환경에 기초한 리덕션 방법(environment based reduction)"을 사용하였다는 것이다. 이와 같은 접근 방법을 사용한 이유는 논리 언어를 위한 WAM이라는 강력한 추상 머신이 이미 존재하고 있기 때문이며, 또한 WAM에서는 환경을 이용하여 논리 절에 있는 변수를 바인딩 하기 때문에 환경에 바탕을 둔 리덕션 방법을 사용하였다. 이러한 이유 때문에 F-WAM에서는 그래프 리덕션(graph reduction) 방법 보다 지연 계산(lazy evaluation)을 효율적으로 할 수 없지만, 빠른 리덕션을 할 수 있다는 장점을 가지고 있다.

제 5장은 다음과 같이 구성되어 있다. 5.2절에서는 함수 논리 언어의 일종인 Aflog에 대하여 설명하고, 5.3절에서는 Aflog에서 사용되는 추론 방법을 제공하기 위한 F-WAM의 기능에 대하여 설명하도록 한다. 5.4절에서는 제안한 추상 머신 F-WAM에 대한 구조와 인스트럭션 집합에 대하여 설명하고, 5.5절에서는 F-WAM에 대한 시뮬레이션과 성능 평가에 대하여 설명하도록 한다. 5.6절에서는 관련된 연구들에 대한 비교를 하며, 마지막으로 5.7절에서는 결론을 기술하였다.

## 5.2 함수 논리 언어 Aflog와 Canonical Unification

Aflog[Shin87, Shin88]는 서론에서 언급한 함수 논리 언어 중에서 두번째 부류에 속하는 언어로서, 같음-단일화 알고리즘으로 Canonical Unification을 사용한다. 본 절에서는 Aflog에 대한 구문과 수행의미(operational semantics), 그리고 Canonical Unification에 대하여 예제를 통하여 간략하게 알아보도록 한다. 이 언어에 대한 내용은 [Shin87]과 [Shin88]에 자세히 설명되어 있다.

Aflog 프로그램은 함수에 대한 정의인 등식(equation)과 논리 언어의 일종인 Prolog 논리 절로 이루어져 있는데, 함수에 대한 정의인 등식은 다음과 같은 구문 형식을 갖는다. 여기서 '==>'는 등호를 나타내는 기호이다.

```
A(t1,...,tn) ==> (G1 | B1),
                 (G2 | B2),
                 ....
                 (Bn)
```

위의 형식에서 t1,..,tn은 데이타 항(data term)이며, G1,...,Gn은 boolean 수식을 나타내고, B1,..,Bn은 일반적인 항이다. '|'는 guard 명령으로서 Concurrent Prolog에서 사용하는 guard와 같은 의미를 가지고 있다. 즉, A라는 함수 항은 패턴(t1,..,tn)이 일치가 된 다음에

G1이 만족될 경우 B1과 같고, G2가 만족될 경우에는 B2와 같으며, 모든 guard가 전부 만족하지 않으면 Bn과 같다는 의미를 가지고 있다.

Canonical Unification은 제한된 같음-단일화 알고리즘으로서, 같음-단일화 알고리즘의 효율성을 위하여 함수 논리 언어에서 사용하는 등식에 다음과 같은 제약을 가하였다. 이 제약은 비록 함수 논리 언어의 표현력을 감소시키지만, Canonical-Unification의 효과적인 수행에 중요한 역할을 한다.

가) 등식은 canonical하다는 성격을 가져야 한다. 등식의 집합 E가, 모든 항 M, N, P에 대하여 P가 M으로 대치되고 (P -->* M) P가 N으로 대치될 때 (P -->* N), M -->* Q 이고 N -->* Q 되는 Q가 존재할 때, 등식의 집합 E는 confluent하다고 하며, 임의의 항 M에 대해서 M --> M1 --> M2 -->... 되는 무한 대치 과정이 없을 때 등식의 집합 E는 terminating하는 성질을 가졌다고 한다. 이때 confluent 성질과 terminating 성질을 만족할 때 등식의 집합은 canonical하다고 한다.

나) 함수 항은 reduction될 때 논리 변수를 가져서는 안된다. 즉 함수의 모든 인수들은 reduction될 때 항상 ground 상태로 있어야 된다.

주어진 등식이 위와 같은 두가지 조건을 만족하면 함수 응용의 값, 즉 함수 항의 정규형은 오직 하나의 정규형으로 결정된다. 이와 같은 성질을 이용하여 Canonical Unification에서는 다음과 같은 두 단계를 거쳐서 두 항을 단일화한다. 먼저 두 항을 주어진 등식을 이용하여 더 이상 적용할 등식이 없을 때까지 정규화(normalizing, reduction, rewriting)하여 두 항을 모두 완전한 정규형으로 만든다. 다음에는 이 정규형을 가지고 보통의 Prolog에서 사용하는 단일화(즉, 구문적 단일화 : 두 항이 unify될 수 있는가를 조사하는 과정)을 수행한다. 알고리즘 1)은 위에서 설명한 Canonical Unification 알고리즘을 나타낸 것이다.

---

알고리즘 1) : Canonical-Unification 알고리즘

입력 : 항 T 와 S.
출력: mgu $\sigma = \{ x_1/m_1, x_2/m_2, .., x_p/m_p \}$ 혹은 단일화할 수 없다는 메시지

1. T와 S를 정규화 시킨다
   Tn = complete-rewriting(T)
   Sn = complete-rewriting(S)
2. Tn과 Sn을 단일화 시킨다. Tn과 Sn이 단일화된다면 mgu $\sigma = \{ x_1/m_1, x_2/m_2, .., x_p/m_p \}$를 출력하고 그렇지 않을 경우에는 단일화할 수 없다는 메시지를 출력함.

---

Aflog의 Canonical Unification은 완전성(completeness)이 증명되어 있으며, Canonical Unification이 가지고 있는 제약점, 즉 등식이 canonical 하여야 한다는 것과 함수 항의 인수로 unbound된 변수가 나타나서는 않된다는 규칙을 어기지 않고도 여러 알고리즘을 쉽게 프로그래밍할 수 있는 것으로 알려져 있다[Shin87].

예제 5.1)과 예제 5.2)는 Aflog로 작성된 함수 논리 언어 프로그램으로서, 예제 5.1)은 두 개의 입력 리스트를 받아서 하나의 출력 리스트를 출력하는 프로그램이며, 예제 5.2)는 리스트를 입력으로 받아서 정렬된 이진 트리를 출력하는 프로그램이다. 이런 예제에서 보는 것과 같이 한 프로그램안에 논리 언어 형태의 프로그램과 함수 언어 형태의 프로그램이 공존할 수 있기 때문에 좀 더 효율적인 프로그램을 작성할 수 있다.

예제 5.1) append 프로그램

```
append([],Y) => Y.
append([H|T],Y) => [H|append(T,Y)].
```

예제 5.2) Make_Binary_Tree 프로그램

```
    /* 정렬 이진 나무에 새로운 원소를 첨가 시키는 등식 */
insert(A,empty)          ==> tree(empty,A,empty).
insert(A, tree(L,B,R)) ==> ((A<B) | tree(insert(A,L),B,R)),
                           (tree(L,B, insert(A,R))).

    /* 논리적인 정의 */
make-binary-tree(L,Tr)  :-  build_up(L,empty,Tr).

    /* 주어진 리스트를 가지고 sorted 이진 나무를 만드는 논리 절 */
build_up([],Tr,Tr).
build_up([A|L],Tr,NewTr) :- build_up(L, insert(A,Tr),NewTr)
```

## 5.3. F-WAM에서의 추론 방법

서론에서 언급한 것과 같이 F-WAM의 추론 방법은 SLD-resolution과 환경에 기초한 리덕션이다. 그런데 환경에 기초한 리덕션 방법에서는 outermost 리덕션 보다는 innermost 리덕션을 효율적으로 구현할 수 있기 때문에, innermost 리덕션 방법을 사용하여 함수 논리 언어의 함수 언어 부분을 수행한다.

F-WAM에서의 추론 방법을 좀 더 자세히 알아보면 다음과 같다. F-WAM에서 어떤 논리절의 서브 고울이 인수로 함수 응용 항을 가지고 있는 경우(예를들어 p를 술어(predicate) 이름, f를 함수 이름이라 할 때 p(f(a),b)인 경우)에는, 앞에서 언급한 것과 같이 우선 함수 응용 항을 innermost 방법으로 정규형을 얻어야 한다. 이때 이런 정규화 과정이 성공하여 함수 응용 항에 대한 정규형을 구할 수 있는 경우에는, 이 정규형을 가지고 WAM의 단일

화 명령어를 수행하면 된다. 그런데 만약 이런 정규화 과정이 성공하지 못할 경우를 가정하여 보자. 이런 경우는 함수 응용 항이 인수로 논리적 변수(logical variable)를 가지고 있거나 (예를 들어 p(f(X),a)에서 정규화 과정을 수행할 때 X가 논리적 변수인 경우), 함수 응용의 인수 패턴에 맞는 함수의 정의가 없을 경우인데, 이때는 WAM의 'fail' 동작을 수행하여 앞의 서브 고울을 다시 수행한다. 이 추론 과정을 예를들어 설명하면 다음과 같다.

예제 5.3)

```
p(X,Y) :- q(X,Z), r(factorial(Z),Y).

factorial(Z) ==> (Z == 1) | 1,
                 (Z > 1)  | Z * factorial(Z-1).
```

- 경우 1)

  만약 q가 Z를 어떤 정수(여기서는 Z=3이라고 가정하자)로 바인딩하였다면 r(factorial(Z),Y)에 대한 Canonical Unification 과정에서는, 먼저 factorial(Z)에 대한 정규형을 구한다. 즉, 앞에서 가정한 것과 같이 Z를 3이라 하면 factorial(3)의 정규형은 6이 되어, 결과적으로 F-WAM은 r(6,Y)를 WAM과 같은 방법으로 수행하면 된다. 만약 r(6,Y)가 실패하거나 p(X,Y) 이후에 있는 서브 고울이 실패하면, WAM과 같은 'fail' 동작을 수행한다. 즉, recent-choice-point가 포인트하고 있는 서브 고울을 다시 수행하게 된다. 그런데 Aflog에서의 함수 정의는 canonical 특징을 가지고 있기 때문에 factorial(3)은 6이외에 다른 정규형이 없다. 그러므로 정규화 과정을 다시 수행할 필요가 없다. 즉, factorial 함수에 대한 choice-point는 필요 없게 된다. 여기서 q(X,Z)가 recent-choice-point가 포인트하고 있는 서브 고울이라고 가정하면, q(X,Z)가 다시 수행되어 다른 Z값을 생성하게 된다.

- 경우 2)

  만약 q(X,Z)가 -1과 같은 값을 Z에 바인딩하게 되면, 앞에서 언급한 Canonical Unification의 함수 정규화 과정에서 factorial(-1)에 맞는 함수 정의를 찾을 수 없기 때문에, F-WAM은 'fail' 동작을 수행한다.

- 경우 3)

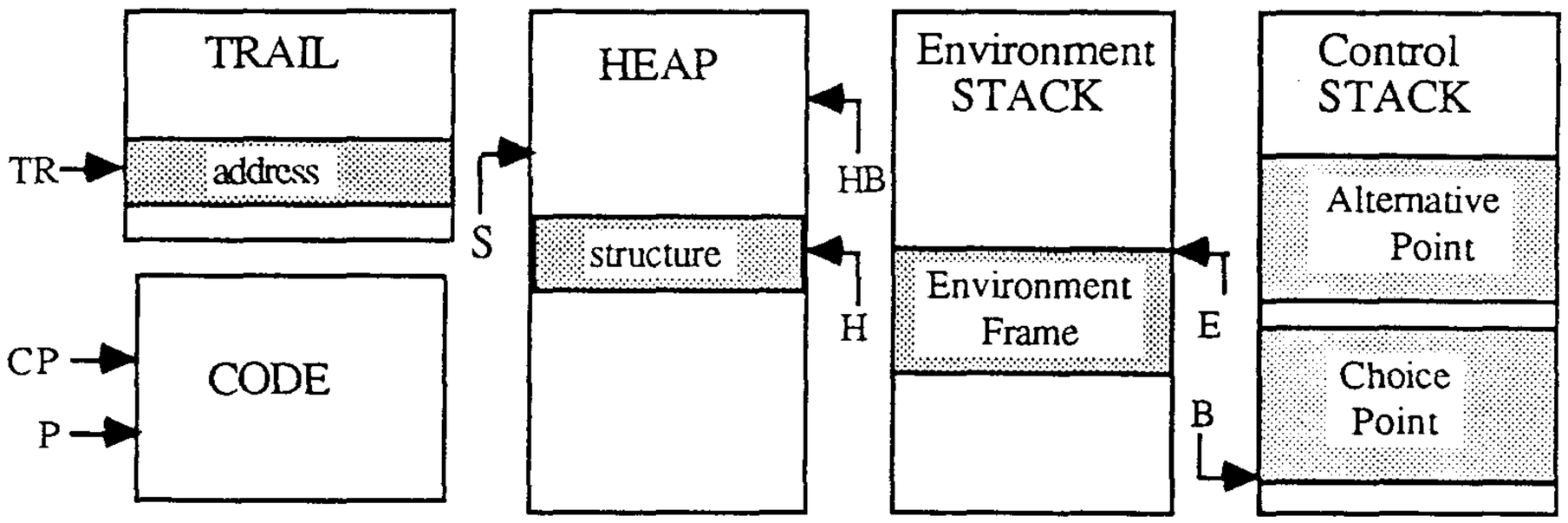
  만약 q(X,Z)가 Z의 값을 바인딩하지 않으면, 즉, factorial(Z)를 정규화할 때 Z가 논리적 변수로 남아있는 경우에는 경우 2)와 마찬가지로 F-WAM은 'fail' 동작을 수행한다. 그러므로 F-WAM의 패턴 매칭 인스트럭션은 함수의 인수로 넘어오는 값이 변수인지를 항상 검사하여야 한다.

## 5.4. F-WAM의 수행시 구조 및 인스트럭션 집합

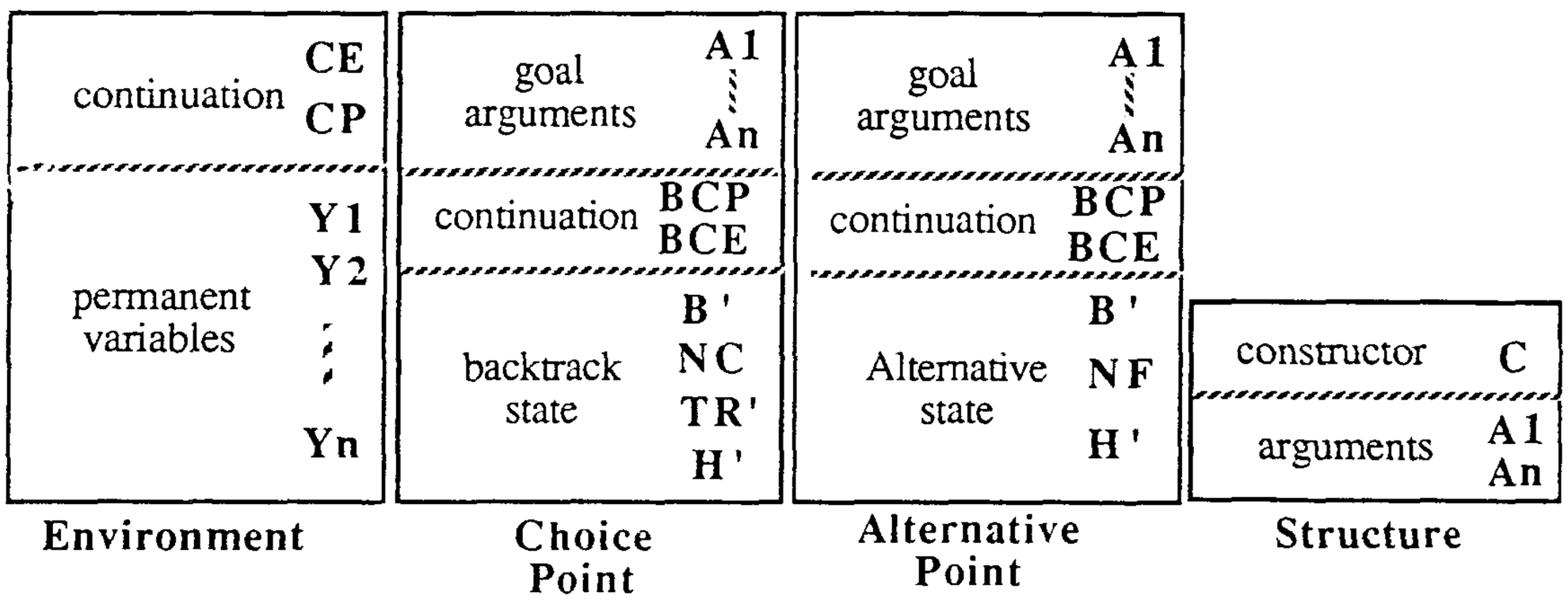
F-WAM은 WAM의 확장형으로서, WAM에 환경에 기초한 리덕션을 할 수 있는 구조 및 인스트럭션을 첨가한 것이다. 본 절에서는 F-WAM의 수행시 데이타 구조와 인스트럭션에 대하여 알아보도록 한다. F-WAM의 수행시 구조는 WAM과 비슷하지만, 패턴 매칭을 위한 인스트럭션과 함수의 리덕션에 관한 인스트럭션을 가지고 있는 것이 WAM과는 다른 점이다.

### 5.4.1 F-WAM의 수행시 구조

앞에서 언급한 것과 같이 F-WAM의 수행시 구조는 논리 언어를 수행하기 위한 WAM 구조에 함수 언어를 수행하기 위한 구조를 포함하고 있다.




(a) F-WAM의 storage 모델




(b) F-WAM의 Run-time structure

<그림 5.1> F-WAM의 수행시 구조

<그림 5.1>에 나타난 것과 같이 F-WAM의 수행시 구조는 WAM의 split-stacking 모델[Tick87a]의 변형으로서, 원래 WAM에서 STACK 영역에 있던 환경과 choice-point를 분리한 구조에 함수 리덕션을 위한 구조를 첨가한 것이다. 메모리 영역은 인스트럭션을 저장하는 코드 영역(Code Area), 구조(structure)와 리스트를 저장하는 Heap 영역, 실행시의 수행 정보를 저장하는 환경 스택(Environment Stack)과 제어 스택(Control Stack), 그리고 단일화를 수행하는 동안 새로이 바인딩되는 변수등을 저장하는 Trail 영역등으로 이루어져 있다. WAM의 레지스터들로는 P(Program Pointer), CP(Continuation Program Pointer), E(Last Environment Pointer), B(Backtrack Pointer), TR(Top of Trail), H(Top of Heap), S (Structure Pointer), HB(Heap Backtrack Pointer), A1,..,An (Argument Registers), X1,.., Xn (Temporary Register) 등이 있는데, 주로 포인터로 사용된다. 이때 Ai 레지스터는 프로시쥬어에 인수를 전달하는데 사용되고, Xi 레지스터는 논리 절의 임시 변수를 저장하는데 주로 사용된다.

WAM에서 환경은 논리 절에 있는 변수들을 위한 기억 장소와, 그 논리 절이 끝났을 때 돌아가야 할 포인트에 대한 정보를 가지고 있는 *continuation point*로 이루어져 있다. 이런 WAM의 환경은 함수 논리 언어의 함수를 리덕션할 때도 똑 같은 방법으로 사용될 수 있다.

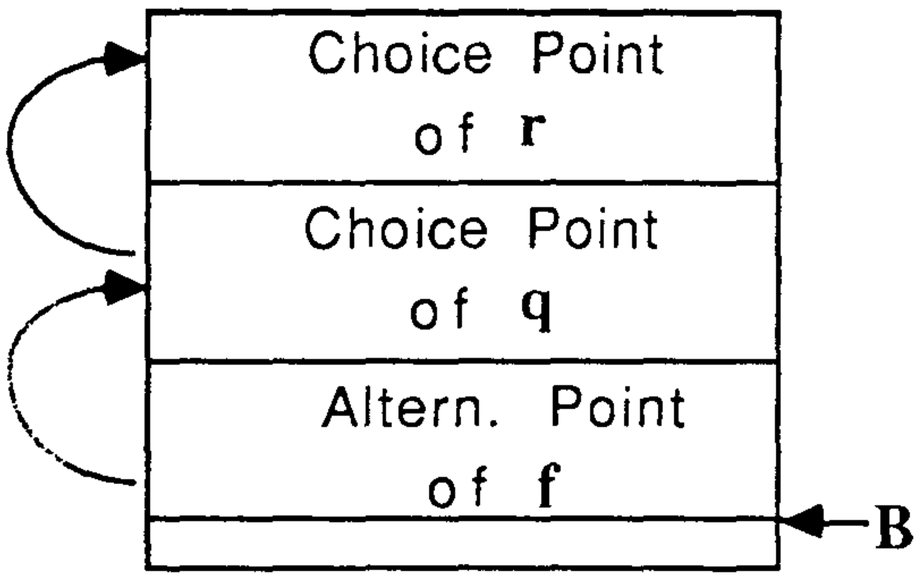
5.2절에서 언급한 것과 같이 하나의 등식이 적용되기 위해서는, 호출시의 입력 패턴과 등식의 헤드 패턴이 부합되어야(match)할 뿐만 아니라, 그 등식의 가드 부분도 만족하여야 한다. 그런데 가드에는 어떠한 부울 수식(boolean expression)이라도 올 수 있기 때문에, 가드를 검사할 때 머신의 상태가 바뀔 수 있다. 그러므로 함수 응용을 리덕션하기 위하여 패턴 매칭을 수행하기 전에 머신의 상태를 기억시켜야 한다. 이런 목적을 위하여 사용되는 F-WAM의 수행시 구조는 *Alternative Point*(AP)인데, 이 구조는 WAM에서 백트래킹을 위하여 사용되는 구조인 choice-point와 비슷한 구조를 가지고 있다. 하지만 함수 언어 부분에서는 백트랙킹이 없기 때문에 AP는 한 등식이 적용될 때 제거될 수 있다. 또한 함수 언어 부분의 패턴 매칭에서는 출력 바인딩(output binding)이 없기 때문에 전의 trail 포인트도 기억할 필요가 없다. 예제 5.4)는 이런 F-WAM의 동작을 설명한 것이다.

예제 5.4)

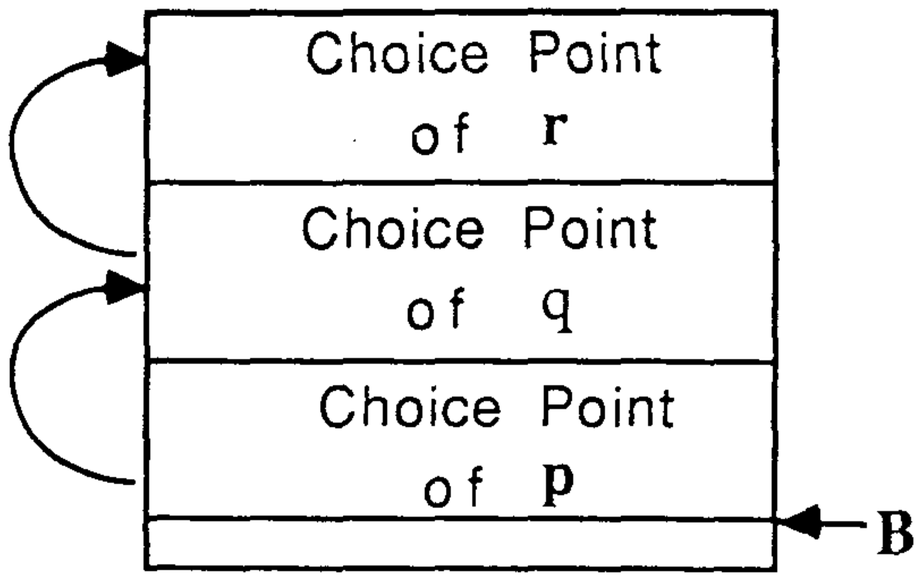
```
r(X) :- q(X), p(f(X), Y), ...   .
r(X).

f(X) ==> (g(X) == 1) ? a,
         (g(X) == 2) ? b,
         (g(X) == 3) ? c.
```

예제 5.4)에서 f(X)의 가드 부분에 있는 g(X)의 수행을 위해서는 머신의 상태를 바꿀

수도 있기 때문에, 패턴 매칭을 수행하기 전에 머신의 상태를 기억시켜야 한다 (그림 5.2-(a)). 이때 f(X)에 가드중의 하나가 성공하면 f(X)에 대한 AP를 제거할 수 있다 (그림 5.2-(b)). 이런 상태에서 p(*f(X)*의 정규형, Y)가 실패하면, recent-choice 포인트가 가르키는 q(X)가 X에 대한 새로운 값을 생성하게 된다.




(a) f(X)가 fire되기 전의
Control Stack




(b) f(X)가 fire된 후의
Control Stack

<그림 5.2> 예제 5.4)에 대한 F-WAM의 상태 변환

### 5.4.2 F-WAM의 인스트럭션 집합

일반적으로 Aflog 프로그램에 있는 하나의 심볼은 F-WAM의 한 인스트럭션과 이들을 묶는 인스트럭션으로 대응된다. <표 5.1>은 F-WAM의 인스트럭션 집합을 나타낸 것이다. F-WAM의 인스트럭션 집합은 WAM 인스트럭션들과 함수 리덕션을 위한 인스트럭션으로 나눌 수 있는데, 논리 언어 부분을 위한 WAM 인스트럭션 부류로는 *Get, Put, Unify, Procedure Control, Indexing,* 그리고 *Clause Control* 인스트럭션이 있다. 이러한 WAM 인스트럭션에 대한 자세한 내용은 [Gabr85]에 설명되어 있기 때문에 여기서는 생략하고, 함수 언어 부분을 위한 리덕션 인스트럭션들만을 설명하기로 한다.

F-WAM에서 함수 리덕션을 위한 인스트럭션 부류로는 *Fget, Rewrite, Fcall,* 그리고 *Commit* 인스트럭션 등이 있다.

- Fget 인스트럭션 부류 :

  이 인스트럭션 부류는 호출할 당시의 인수 패턴이 등식의 헤드 부분 패턴과 일치하는가를 검사하는 인스트럭션으로서, WAM의 Get 인스트럭션과는 달리 인수가 변수인가도 검사하여야 한다. 만약 인수가 변수면 'fail' 동작을 수행하여 앞의 서브 고울에서 다른 인수를 생성하게 한다.

<표 5.1> F-WAM에 대한 인스트럭션 집합

| The Complete F-WAM Instruction Set | | | | | |
|---|---|---|---|---|---|
| **WAM Instructions** | | | | | |
| **Procedure Control** | | **Indexing** | | **Clause Control** | |
| try | L | switch_on_term | v,c,l,s | call | P/arity |
| retry | L | switch_on_constant | n,ff | execute | |
| trust | L | switch_on_structure | n,ff | proceed | |
| try_me_else | L | | | allocate | |
| retry_me_else | L | | | deallocate | |
| trust_me_else | fail | | | | |
| **Get** | | **Put** | | **Unify** | |
| get_variable | Vi,Ai | put_variable | Vi,Ai | unify_variable | Vi |
| get_value | Vi,Ai, | put_value | Vi,Ai | unify_value | Vi |
| | | put_unsafe_value | Yi,Ai | unify_unsafe_value | Yi |
| get_constant | C,Ai | put_constant | C,Ai | unify_constant | C |
| get_list | Ai | put_list | Ai | unify_list | Ai |
| get_structure | Ai | put_structure | Ai | unify_structure | Ai |
| get_nil | Ai | put_nil | Ai | unify_nil | Ai |
| | | | | unify_void | |
| **Reduction Instructions** | | | | | |
| **Fget** | | **Rewrite** | | **Fcall** | |
| fget_value | Vi,Ai | rewrite_value | Ai | fcall | P/arity |
| fget_constant | C,Ai | rewrite_constant | C | | |
| fget_list | Ai | rewrite_nil | | **commit** | |
| fget_structure | Ai | | | commit | |
| fget_nil | Ai | | | | |

- Rewrite 인스트럭션 부류 :

이 인스트럭션 부류는 인수트럭션의 인수가 포인트하고 있는 함수 응용의 정규형을 리턴하는 인스트럭션이다. 예를들어 'rewrite A1'이라는 인스트럭션은 A1에 저장되어 있거나 A1이 포인트하는 구조를 함수 응용의 정규형으로 리턴하는 인스트럭션이다.

- Fcall 인스트럭션 :

Fcall 인스트럭션은 컴파일된 함수 정의(등식)를 호출하는 인스트럭션으로서, 'fcall Proc/Arity Bn'의 형식을 갖는데, 여기서 Bn은 정규형을 포인트하는데 사용되는 레지스터나 변수이다. 그러므로 위의 rewrite 인스트럭션들은 리턴되기 전에 An 값을 Bn에 복사하도록 하여야 한다. 중첩된 함수 응용인 경우에는 가정 안쪽에 있는 함수 호출을 먼저하고, 그

결과를 레지스터나 변수를 통하여 다음 호출에 사용되게 한다. 이런 방법은 innermost 리덕션을 제공하기 위하여 사용되는 방법이다.

Commit 인스트럭션

Commit 인스트럭션은 함수 언어의 패턴 매칭을 위한 인스트럭션이다. 4.1절에서 언급한 것과 같이 AP는 함수 언어 부분의 패턴 매칭을 수행할 때 생성되는 데이타 구조로서 한 등식의 패턴 매칭과 가드가 성공하면 제거할 수 있는 데이타 구조이다. 이런 작업을 하여 주는 인스트럭션이 commit 인스트럭션이다. 그러므로 이 인스트럭션은 패턴 매칭에 대한 인스트럭션과 가드를 검사하는 인스트럭션 다음에 나오는 인스트럭션이다.

Aflog 프로그램으로 부터 F-WAM 코드가 어떻게 생성되는 가를 다음 예제를 통하여 알아보자. <그림 5.3>은 앞에서 설명한 Aflog append 프로그램(<그림 5.3>-a)과 컴파일된 F-WAM 코드(<그림 5.3>-b)를 나타낸 것이다. 여기서 'Ai'는 F-WAM의 레지스터를 나타낸 것인데, F-WAM에서 레지스터는 인수 전달과 중간 결과를 저장하는데 사용된다. <그림 5.3>-b에서 옆에 나와있는 Aflog 심볼들은 그 인스트럭션이 어떻게 추출되었는 가를 나타낸 것이다.

```
F1 : append([],X) ==> X.
F2 : append([X|U],V) ==> [X|append(U,V)].
```

(a) Aflog로 작성된 append 프로그램

```
append : switch_on_term  A1, fail, F1, F2, fail

F1     : fget_nil        A1              % append([],X)
         rewrite_value   A2              %              ==> X
         proceed                         %

F2     : allocate
         fget_list       A1              % append([
         unify_value     'X'             %          X
         unify_value     'U'             %            |U],
         fget_value      'V', A2         %               V)
         put_value       'U', A1         % append(U,
         put_value       'V', A2         %            V
         fcall           append/2, A2    %          )
         put_list        A3              % ==> ( [
         unify_value     'X'             %        X
         unify_value     A2              %          |append(U,V)]
         rewrite_value   A3              %      )
         deallocate
         proceed                         %      .
```

(b) (a)의 append 프로그램에 대한 컴파일된 F-WAM 코드

<그림 5.3> Aflog로 작성된 append 프로그램과 컴파일된 F-WAM 코드

<그림 5.3>에 나와있는 F-WAM 코드는 비록 되돌림(recusive) 특징을 가지고 있지만, WAM과 같이 tail-recursion 특징을 가지고 있지는 못하다. 그 이유는 모든 함수가 계산이 되어야지만 그 함수에 대한 환경을 제거할 수 있기 때문이다. append를 정의한 두 등식은 첫번째 인수에 의하여 구별할 수 있기 때문에 AP 구조는 필요없게 된다. 이 코드에 대한 또 다른 특징으로는 F2에서 첫번째 인수의 헤드 부분을 환경에 저장하는 것인데, 이 값은 테일 부분의 함수 응용이 리턴될 때까지 저장되게 된다. put_list 인스트럭션은 이 두 값을 연결하는데 사용되는데, 이 리스트가 append 함수 응용에 대한 정규형으로서 rewrite 인스트럭션과 proceed 인스트럭션에 의하여 호출한 프로시쥬어로 리턴되게 된다.

두 번째 예제는 앞에서 설명한 Aflog로 작성된 make-binary-tree 프로그램 중에서 insert 함수에 대한 Aflog 프로그램과 그 컴파일된 F-WAM 코드이다.

```
F1 : insert(A, empty) ==> tree(empty, A, empty).
F2 : insert(A, tree(L,B,R)) ==>
                (A<B) | (tree(insert(A, L), B, R)),
F3 :            (A>=B) | (tree(L, B, insert(A, R))).
```

(a) Aflog로 작성된 insert 함수

```
        procedure     insert
insert: switch_on_term A2, fail, F1, fail, F2
F1    : fget_value_XA   A1,A1          % insert(A,
        fget_constant   'empty', A2    %          empty)
        put_structure   'tree', A2     % tree(
        unify_constant  'empty'        %      empty,
        unify_value     A1             %      A,
        unify_constant  'empty'        %      empty
        rewrite_value   A2             %    )
        proceed                        %      .
F2    : try_me_else     F3
        allocate
        get_value       'A', A1        % insert(A,
        fget_structure  'tree', A2     %          tree(
        unify_value     'L'            %               L,
        unify_value     'B'            %               B,
        unify_value     'R'            %               R))
        put_value       'A', A1        % (A<
        put_value       'B', A2        %  B
        lt A1, A2                      % )
        commit                         % |
        put_value       'A', A1        % insert(A,
        put_value       'L', A2        %          L
        fcall           insert/2,A1    %        )
        put_structure   'tree', A2     % tree(
        unify_value     A1             %      insert(A,L),
```

```
        unify_value     'B'             %       B,
        unify_value     'R'             %       R
        rewrite_value   A2              %     )
        deallocate
        proceed                         %     .
F3 :    trust_me_else_fail
        allocate
        get_value       'A', A1         % insert(A,
        fget_structure  'tree', A2      %        tree(
        unify_value     'L'             %             L,
        unify_value     'B'             %             B,
        unify_value     'R'             %             R))
        put_value       'A', A1         % (A>=
        put_value       'B', A2         %  B
        ge A1, A2                       % ) ?
        put_value       'A', A1         % insert(A,
        put_value       'R', A2         %        R
        fcall           insert/2,A1     %       ).
        put_structure   'tree', A2      % tree(
        unify_value     'L'             %      L,
        unify_value     'B'             %      B,
        unify_value     A1              %      insert(A,R)
        rewrite_value   A2              %    )
        deallocate
        proceed                         %   .
```

(b) (a)의 insert 함수에 대한 컴파일된 F-WAM 코드

<그림 5.4> Aflog로 작성된 insert 함수와 컴파일된 F-WAM 코드

<그림 5.4>에 있는 insert 함수는 <그림 5.3>에 있는 append 프로그램과는 달리 인수만으로 적용할 등식을 구별할 수 없기 때문에 AP 구조가 필요하며, 이 AP는 try_me_else 인스트럭션에 의하여 생성되고, 패턴 매칭 인스트럭션과 가드를 검사하는 인스트럭션 다음에 있는 commit 인스트럭션이나 trust_me_else 인스트럭션에 의하여 제거된다.

## 5.5 시뮬레이션과 결과 분석

본 연구에서 제안한 F-WAM에 대한 성능을 평가하기 위하여 F-WAM에 대한 시뮬레이터를 C 언어를 이용하여 SUN3/160에서 작성하였다. 본 장에서는 F-WAM에 대한 시뮬레이션과 그 분석에 대하여 알아보도록 한다.

### 5.5.1 성능 평가 기준과 가정

F-WAM의 성능은 여러 벤치마크 프로그램을 수행하는데 걸리는 시간과 필요한 메모리 양을 기준으로 측정하였으며, 그 결과를 같은 의미를 갖는 Prolog 프로그램을 수행한 WAM의 결과와 비교하였다. 예상되는 수행 속도는 컴파일된 벤치마크 프로그램을 F-WAM에서 수행

하는데 필요한 메모리 참조횟수와 레지스터 전송횟수를 기준으로 하였는데, 그 속도 비율은 3 : 1로 가정하였다. 즉, 메모리를 참조하는데 걸리는 시간은 레지스터 사이 전송 시간의 3배라고 가정하였다. 수행에 필요한 메모리 양은 사용된 Heap 영역과 환경, Choice Point, AP, 그리고 Trail 영역의 합으로 계산하였다.

### 5.5.2 시뮬레이션과 결과 분석

첫번째 벤치마크 프로그램은 앞의 예제에서 언급한 *append* 프로그램이다. Aflog로 작성된 append 프로그램과 Prolog로 작성된 append 프로그램을 각각 컴파일시켜서 F-WAM과 WAM에서 수행시켰을때, 예상되는 수행시간과 필요한 메모리 양을 <표 5.2>에 나타내었다.

<표 5.2> *append* 프로그램에 대한 WAM과 F-WAM의 성능

(a) *append*를 수행할 때 F-WAM의 성능

| 수행시 성능 / 리스트의 갯수 | 메모리 사용량 | | | | 수행속도에 관련된 요인 | | |
|---|---|---|---|---|---|---|---|
| | Environment 스택 사용량 | Control 스택 사용량 | Heap 영역 사용량 | Trail 영역 사용량 | 메모리 참조 횟수 | 레지스터 전송 횟수 | 레지스터 비교 횟수 |
| 20 | 101 | 0 | 138 | 0 | 967 | 385 | 380 |
| 40 | 201 | 0 | 258 | 0 | 1887 | 745 | 740 |
| 60 | 301 | 0 | 378 | 0 | 2807 | 1105 | 1100 |
| 80 | 401 | 0 | 498 | 0 | 3727 | 1465 | 1460 |
| 100 | 501 | 0 | 618 | 0 | 4647 | 1825 | 1820 |

(b) *append*를 수행할 때 WAM의 성능

| 수행시 성능 / 리스트의 갯수 | 메모리 사용량 | | | | 수행속도에 관련된 요인 | | |
|---|---|---|---|---|---|---|---|
| | Environment 스택 사용량 | Control 스택 사용량 | Heap 영역 사용량 | Trail 영역 사용량 | 메모리 참조 횟수 | 레지스터 전송 횟수 | 레지스터 비교 횟수 |
| 20 | 6 | 0 | 138 | 1 | 957 | 472 | 448 |
| 40 | 6 | 0 | 258 | 1 | 1857 | 912 | 868 |
| 60 | 6 | 0 | 378 | 1 | 2757 | 1352 | 1288 |
| 80 | 6 | 0 | 498 | 1 | 3657 | 1792 | 1708 |
| 100 | 6 | 0 | 618 | 1 | 4557 | 2232 | 2128 |

(c) WAM과 F-WAM의 비교

- 총메모리 사용량 = Environment_Stack + Control_Stack + Heap + Trail
- 예상 수행 속도 = (총 메모리 참조 횟수 x 3) + 총 레지스터 동작 횟수

| 수행시 성능 / 리스트의 갯수 | F-WAM 총 메모리 사용량 | F-WAM 예상 수행 시간 | WAM 총 메모리 사용량 | WAM 예상 수행 시간 |
|---|---|---|---|---|
| 20 | 239 | 3666 | 145 | 3791 |
| 40 | 459 | 7146 | 265 | 7351 |
| 60 | 679 | 10626 | 385 | 10911 |
| 80 | 899 | 14106 | 505 | 14471 |
| 100 | 1119 | 17586 | 625 | 18031 |

<표 5.2>에서 볼 수 있는 것과 같이 append 프로그램을 수행하는데 필요한 수행 시간은 F-WAM이나 WAM 경우 거의 비슷하다. 하지만 F-WAM의 경우 더 많은 메모리 영역을 사용함을 알 수 있다. 그 이유는 WAM의 경우에는 마지막 고울을 수행하기 전에 환경을 회수하는 tail-recursion-optimization을 적용할 수 있지만, F-WAM의 경우에는 마지막 함수 호출의 결과를 받아서 리턴해야 하기 때문에 이 방법을 사용할 수 없다. 따라서 같은 프로그램을 수행하는데 F-WAM이 더 많은 메모리 영역을 필요로 한다. 이와같은 append 프로그램은 F-WAM의 단점을 보여주는 것이다. 하지만 이런 단점은 다음 벤치마크 프로그램을 통하여 알 수 있는 것과 같이 F-WAM의 다른 장점으로 인하여 상쇄될 수 있다.

다음 벤치마크 프로그램은 *make binary tree* 프로그램이다. 앞에서와 마찬가지로 이 프로그램을 Aflog로 작성한 경우와 Prolog로 작성한 경우를 각각 F-WAM과 WAM으로 컴파일하여 수행시켰을 때의 각각의 성능을 <표 5.3>에 나타내었다.

<표 5.3> *make binary tree* 프로그램에 대한 WAM과 F-WAM의 성능

(a) *make-binary-tree*를 수행할 때 F-WAM의 성능

| 수행시 성능 / 리스트의 갯수 | 메모리 사용량 | | | | 수행속도에 관련된 요인 | | |
|---|---|---|---|---|---|---|---|
| | Environment 스택 사용량 | Control 스택 사용량 | Heap 영역 사용량 | Trail 영역 사용량 | 메모리 참조 횟수 | 레지스터 전송 횟수 | 레지스터 비교 횟수 |
| 20 | 68 | 9 | 540 | 1 | 11537 | 4103 | 4125 |
| 40 | 128 | 9 | 1880 | 1 | 43627 | 15373 | 15625 |
| 60 | 188 | 9 | 4020 | 1 | 96317 | 33843 | 34525 |
| 80 | 248 | 9 | 6960 | 1 | 169607 | 59513 | 60825 |
| 100 | 308 | 9 | 10700 | 1 | 263497 | 92383 | 94525 |

(b) *make-binary-tree*를 수행할 때 WAM의 성능

| 수행시 성능 / 리스트의 갯수 | 메모리 사용량 | | | | 수행속도에 관련된 요인 | | |
|---|---|---|---|---|---|---|---|
| | Environment 스택 사용량 | Control 스택 사용량 | Heap 영역 사용량 | Trail 영역 사용량 | 메모리 참조 횟수 | 레지스터 전송 횟수 | 레지스터 비교 횟수 |
| 20 | 108 | 300 | 580 | 112 | 14241 | 5913 | 6974 |
| 40 | 208 | 600 | 1960 | 422 | 52241 | 21593 | 25724 |
| 60 | 308 | 900 | 4140 | 932 | 114041 | 47073 | 56274 |
| 80 | 408 | 1200 | 7120 | 1642 | 199641 | 82353 | 98624 |
| 100 | 508 | 1500 | 10900 | 2552 | 309041 | 127433 | 152774 |

(c) WAM과 F-WAM의 비교

- 총메모리 사용량 = Environment_Stack + Control_Stack + Heap + Trail
- 예상 수행 속도 = (총 메모리 참조 횟수 x 3) + 총 레지스터 동작 횟수

| 수행시 성능 / 리스트의 갯수 | F-WAM | | WAM | |
|---|---|---|---|---|
| | 총 메모리 사용량 | 예상 수행 시간 | 총 메모리 사용량 | 예상 수행 시간 |
| 20 | 618 | 42839 | 1100 | 55610 |
| 40 | 2018 | 161879 | 3190 | 204040 |
| 60 | 4218 | 357319 | 6280 | 445470 |
| 80 | 7218 | 629159 | 10370 | 779900 |
| 100 | 11018 | 977399 | 15460 | 1207330 |

위의 *make-binary-tree* 경우의 시뮬레이션 결과를 통하여 F-WAM이 WAM 보다 빠르게 수행할 수 있음을 알 수 있었으며, 또한 F-WAM이 WAM의 경우 보다 메모리를 더 적게 사용함을 알 수 있다. 이와같이 F-WAM이 WAM보다 효율적인 이유는 F-WAM의 리덕션 인스트럭션이 대응되는 WAM의 인스트럭션 보다 효율적이기 때문이다. 즉, F-WAM의 patter-matching 인스트럭션이 WAM의 단일화 인스트럭션보다 효율적이기 때문이다. 또한 앞에서 언급한 것과 같이 F-WAM의 AP 구조는 대응되는 WAM의 Choice-Point 구조와는 달리 한 등식이 fire 되면 계속 메모리 영역에 남아있을 필요가 없다. 그렇기 때문에 일반적으로 F-WAM이 비록 tail-recursion-optimization을 제공하지 못하지만, AP의 빠른 제거로 인하여 WAM 보다 적은 메모리 영역을 필요로 한다.

## 5.6 관련된 연구들과의 비교

Prolog나 LISP과 같이 기호처리용(symbolic procerssing) 언어를 위한 머신에 관한 연구는 많이 진행되었으나, 두 언어를 모두 수행시킬 수 있는 머신에 대한 연구는 비교적 적은

상태이다. 논리 언어와 함수 언어를 모두 수행시킬 수 있는 머신에 관한 연구는 Xenologic사의 X-1[Dob87]과 CSELT 연구소의 Levi의 연구[Levi87] 등이 있다. 이들과 F-WAM을 비교하면 다음과 같다.

Xenologic사의 X-1은 기호처리용 언어(특히 Prolog와 LISP)를 위하여 특별히 설계된 머신이다. 이 머신의 구조는 F-WAM과 거의 비슷하며, LISP 프로그램을 WAM 코드로 컴파일하여 수행하는 방법을 사용한다. 그러나 X-1은 함수 논리 언어를 위한 머신이 아니라, Prolog와 LISP을 독립적으로 수행하기 위한 머신이다. 그렇기 때문에 이 머신은 함수 언어와 논리 언어를 각각 독립적으로 수행할 수 있지만, F-WAM과 같이 한 프로그램안서 두 가지 스타일의 언어를 동시에 수행할 수 없다.

CSELT 연구소의 Levi의 방법은 K-LEAF라는 함수 논리 언어를 flattening 방법과 outermost SLD-resolution을 사용하여 WAM의 확장형에서 수행을 한다. 즉, 함수 논리 언어 프로그램의 함수 언어 부분을 논리 언어의 고울로 변환시켜서 수행을 한다. 그렇기 때문에 이 방법에서는 진정한 의미의 함수 리덕션을 위한 메카니즘은 없다고 할 수 있다.

## 5.7 결론 및 앞으로의 연구 방향

함수 논리 언어는 논리 언어의 특징과 함수 언어의 특징을 모두 가지고 있는 매우 강력한 언어이다. 하지만 이런 언어를 효율적으로 수행할 수 있는 방법이 아직 개발되지 않았기 때문에 널리 사용되지 못하고 있다. 이 문제 점을 해결하기 위하여 본 연구에서는 함수 논리 언어의 수행에 필요한 기본 기능을 머신 레벨에서 제공하는 추상 머신을 설계하고, 그 인스트럭션 집합을 제안하였다.

제안된 추상 머신 F-WAM은 Canonical Unification에 바탕을 둔 함수 논리 언어(특히 Aflog)를 효율적으로 수행할 수 있도록 설계되었는데, 설계 방법은 논리 언어 머신인 WAM에 리덕션 방법을 첨가하였다. F-WAM은 같은 기능을 갖는 논리 언어 프로그램을 수행시키는 WAM보다 함수 논리 언어 프로그램을 효율적으로 처리할 수 있음을 시뮬레이션을 통하여 보였다.

하지만 현재의 F-WAM은 함수 언어의 가장 큰 특징 중의 하나인 *lazy-evaluation*을 할 수 없기 때문에 무한 자료 구조를 제공하지 못한다는 단점을 가지고 있다. 그러므로 제안된 F-WAM에 lazy-evaluation을 할 수 있는 방법에 관한 연구가 더 있어야 하겠다.

# 제 6 장 결론 및 앞으로의 연구 방향

본 연구에서는 인공 지능 언어로 많이 사용되고 있는 논리 언어를 병렬로 수행시키기 위한 병렬 추론 머신 X-WAM의 설계 및 구현을 목표로 하고 있다. 본 연구의 선행 연구로서 이미 1,2차년도에 X-WAM의 기본이 되는 병렬 수행모델 XWAM-I과 XWAM-II를 설계 하였고 그 시제품을 일부 구현한 바 있다. 당해년도에는 전년도에 연구된 AND 병렬성을 추구한 XWAM-I과 OR 병렬성을 추구한 XWAM-II를 통합하여 새로운 병렬 수행 모델 X-WAM을 설계하고 그 성능을 시뮬레이션을 통하여 평가하였으며, 그 일부분을 실제 구현하여 프로토타입 병렬 추론 머신을 제작하였다. 또한 실제 사용할 수 있는 시스템으로 만들기 위하여 사용자를 위한 개발 환경을 개발하였으며, X-WAM의 전용 프로세서로 사용할 수 있는 Prolog 전용 프로세서도 설계하였다. 그리고 이런 연구와는 별도로 논리 언어의 표현력을 증가시키기 위하여 논리 언어에 함수 언어의 특징을 첨가시킨 함수 논리 언어 Aflog에 대한 추상 머신도 설계하였다.

현재 구현된 X-WAM은 MC68000을 기본으로 하는 프로세서 8개와 이를 제어하는 호스트 머신으로 구성되어 있다. 현재 이 시스템은 X-WAM의 OR 병렬성만을 이용하도록 되어있는데, OR 병렬성이 많은 응용 프로그램인 경우 성능이 거의 선형적으로 증가한다. 또한 X-WAM의 기본 프로세서 모듈에는 다른 프로세서 모듈과 통신을 할 수 있는 기능을 제공하고 있기 때문에 AND 병렬성도 쉽게 구현할 수 있다. X-WAM의 호스트 머신으로는 386 PC를 사용하였으며, 컴파일러를 비롯한 여러 유틸리티를 포함하고 있는 사용자 인터페이스를 윈도우 시스템을 바탕으로 하여 구현하였다. 실제로 지식 표현 시스템인 Sphinx를 X-WAM에서 수행시킴으로서 실제 사용할 수 있다는 것을 보였다.

앞으로 더 연구해야 할 과제로는 다음과 같은 것들이 있다. 첫째로는 프로세서 보드의 통신 기능을 이용하여 AND 병렬성도 추구하는 완전한 X-WAM 시스템을 구현하는 일이며, 둘째로는 Prolog 전용 프로세서를 구현하여 X-WAM의 프로세서 보드로 사용하여 성능을 높이는 것이다. 세째로는 side-effect를 병렬로 처리할 수 있는 방법을 개발하는 것인데, 이것은 선진 외국의 여러 연구기관에서도 아직 해결하지 못하고 있는 문제이다. 따라서 이런 side-effect를 병렬 환경에서 처리하는 방법을 개발하여 Sphinx와 같은 인공 지능 시스템을 병렬로 수행시키고, X-WAM의 프로세서 갯수를 수십개로 늘려서 성능을 보다 향상시키는 것이 앞으로 수행해야 할 과제이다.

# 참고 문헌


[Barb86] R. Barbuti, M. Bellia, G. Levi, and Martelli, "LEAF: A Language which integrates Logic, Equations and Functions," *LOGIC PROGRAMMING Functions, Relations, and Equations*, D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986.

[Bell86] M. Bellia and G. Levi, "The Relation between Logic and Functional Languages : A Survey," *Journal of Logic Programming 3*, 1986.

[Bell87] M. Bellia, P. G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi, "A two level approach to logic plus functional programming integration," *Proc. 87 PARLE Conference*, Springer-Verlag, 1987.

[Bor84] P. Borgwardt, "Parallel Prolog using Stack Segments on Shared Memory Multiprocessor," *Symposium on Logic Programming*, Feb. 1984.

[Bosc86] P. G. Bosco and E. Giovannetti, "IDEAL : An Ideal Deductive Applicative Language," *Proc. 1986 Symposium on Logic programming*, IEEE Computer Society Press, 1986.

[Carl86] M. Carlsson, "Compilation for Trica and its Abstract Machine," Technical Report 35, UPMAIL, Uppsala University, September 1986.

[Cho87] 조 정 완, 맹 승 렬, 추론 컴퓨터 구조 연구, 1986 국책 연구 개발 사업 과제 최종 보고서, 과학기술처 1987.

[Cho88] 조 정 완, 맹 승 렬, 차세대 컴퓨터 시스템 개발 연구, 1987 국책 연구 개발 사업 최종 보고서, 과학기술처 1988.

[ChD85] J. Chang and A. Despain, "Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis," *Proceedings of the Second Symposium on Logic Programming*, July 1985.

[CiH84] A.Ciepielewski and S.Haridi,"Execution of Bagof on the Or-Parallel Token Machine," Proc. of FGCS'84, pp.551-562, Nov. 1984. *Proceedings of the Forth Symposium on Logic Programming*, August 1987.

[Clo87] W.F. Clocksin, " Principles of the DelPhi Parallel Inference Machine," *The Computer Journal*, vol. 30, No. 5, 1987.

[Con83] J. S. Conery, "The AND/OR Process Model for Parallel Interpretation of Logic Program," *Ph. D. Dissertation*, Dept. of Information and Computer Science, U. C. Irvine, June 1983.

[Con87] J. S. Conery, "Implementin Backward execution in Nondeterministic AND-Parallel Systems," *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Darl86] J. Darlington, A. J. Field, and H. Pull, "The Unification of Functional and Logic Language," *LOGIC PROGRAMMING Functions, Relations, and Equations*, D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986.

[DeG84] D. DeGroot, "Restricted AND-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, Japan, November 1984.

[DeM85] P. Dembinski and J. Maluszynski, "AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs," *Proceedings of the Second Symposium on Logic Programming*, July 1985.

[Dobr85] T. P. Dorby, A. Despain, and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture," *Proc. of 12th Annual International Symposium on Computer Architecture*, IEEE Computer Society, June 1985.

[Dobr87] T. P. Dobry, "A Coprocessor for AI : LISP, Prolog and Data Bases," *Spring Compcon '87*, IEEE Computer Society, 1987.

[Gabr85] J. Gabriel, T. G. Lindholm, E. L. Lusk, and R. A. Overbeek, "A Tutorial on the Warren Abstract Machine for Computational Logic," *Research Paper ANL-84-84*, Argonne National Laboratory, Jun., 1985.

[Gee87] J. Gee, S. W. melvin, Y. N. Patt, "Advantages of Implementing Prolog by Microprogramming a Host General Purpose Computer," *Proc. of Fourth International Conference on Logic Programming*, University of Melborne, MIT Press, May 1987.

[Gogu86] J. A. Goguen and J. Meseguer, "EQLOG: Equality, Types, and Generic Modules for Logic Programming," *LOGIC PROGRAMMING Functions, Relations, and Equations*, D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986.

[GTM84] A.Goto, H.Tanaka, and T.Moto-oka,"Highly Parallel Inference Engine PIE-Goal Rewriting Model and Machine Architecture," *New Generation Computing*, vol.1, 1984.

[Her86a] M. V. Hermenegildo, "An Abstract Machines for Restricted AND-Parallel Execution of Logic Programs," *Proceedings of the Third International Conference on Logic Programming*, July 1986.

[Ichi87] N. Ichiyoshi, T. Miyazaki, and K. Taki, "A Distributed Implementation of Flat GHC on the Multi-PSI," IProc. of the Fourth International Conference on Logic Programming," Melbourne, May 1987.

[ISK84] N.Ito, H.Shimizu, M.Kishi, and K.Rokusawa," Data-Flow Based Execution Mechanisms of Parallel and Concurrent Prolog," *New Generation Computing*, vol.2, 1984.

[Kim86] S.B. Kim, et al., " A Parallel Execution Model of Logic Programs Based on Dependency Relationship Graph," *Int., Conference on Parallel Processing*, Aug., 1986.

[Kim88a] H.C. Kim, "Design of an Abstract Machine for the Goal Process Model," *KAIST CALAB Technical Report*, CAL-TR-010, 1988 (*in Korean*).

[KLP88a] S.B. Kim, H.G.Lee, S.W.Park et al., " A Parallel Execution Model of Logic Programs: Combining the AND/OR Process Model and WAM," (to appear *the Computer Journal*).

[Kim88b] S.B. Kim, et al., " The extended conditional graph expression for restricted AND parallel execution of Prolog programs with intelligent backtracking," KAIST CA-LAB Technical Report, CAL-TR-014 (to appear *the Computer Journal*).

[Kow79] R. Kowalski, *Logic for Problem Solving*, The Computer Science Library, 1979.

[Levi87] G. Levi and P. G. Bosco, "A Complete Semantic Characterization of K-LEAF. A Logic Language with Partial Functions," *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987.

[LeK88] H.G. Lee and S.B. Kim, " X-WAM-II: A static OR parallel execution model for logic programs based on WAM," (submitted to PARLE 89).

[LiM86] P. P. Li and A. J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming," *Proceedings of the Third Symposium on Logic Programming*, September 1986.

[LKL86] Y. J. Lin, V. Kumer, and C. Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," *Proceedings of the Third International Conference on Logic Programming*, July 1986.

[Nakas87] H. Nakashima and K. Nakajima, "Hardware Architecture of the Sequential Inference Machine: PSI-II," *Proc. of 1987 International Symposium on Logic Programming*, IEEE Computer Society, August 1987.

[Nakaz86] Nakazaki R., Konagaya A., Habata S., Shimazu H., Umemura M., and Yamamoto M., "Design of a Co-operative High Performance Sequential Inference Machine (CHI)," *NEC Res. & Develop.*, No.80, January 1986.

[Ove85] R. A. Overbeek et al., " Logic Programming on the HEP," Internal Report, Argone National Lab, U.S.A., 1985.

[Redd86] U. S. Reddy, "Logic Language Based on Functions: Semantics and Implementation," *PhD Dissertation*, Univ. of Utah, Aug., 1986. G. Lindstorm (eds.) Prentice-Hall, 1986.

[Ryu87] K. Y. Ryu, "The Design and Implementation of an Incremental Prolog Compiler," Masters thesis, Computer Architecture Laboratory, KAIST, 1987 (in Korean).

[Shin87] D. W. Shin, J. H. Nang, S. Han, and S. R. Maeng, "A Functional Logic Language Based on Canonical Unification," *Proc. 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987.

[Shin88] D. W. Shin, J. H. Nang, S. R. Maeng, and J. W. Cho, "The Semantics of a Functional Logic Language with Input Mode," *Proc. of the International Conference On the Fifth Generation Computer Systems 1988*, 1988.

[Subr86] P. A. Subrahmanyam and J. H. You, "FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming," *LOGIC PROGRAMMING Functions, Relations, and Equations*, D. DeGroot and G. Lindstorm (eds.) Prentice-Hall, 1986.

[Tick87a] E. Tick, "Studies in Prolog Architecture," Technical Report No. CSL-TR-87-329, Computer Systems Laboratory, Dept. of Electrical Engineering and Computer Science, Stanford University, June 1987.

[Tick87b] E. Tick, "A Prolog Emulator," Technical Note CSL-TN-87-324, Computer Science Laboratory, Stanford University, Stanford, CA 94305, May 1987.

[Turn85] D. A. Turner, "Miranda : A Non-strict Functional Language with Polymorphic Types," *Proc. of the IFIP International Conference on Functional Programming Language and Computer Architecture*, Springer Lecture Notes in CS LNCS201, 1985.

[War77] D. H. D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs," *TR39 & 40*, Dept. of AI, Univ. of Edinburgh, 1977.

[War83] D.H.D. Warren, "An Abstract Prolog Instruction Set," *Technical Report 309*, Artificial Intelligence Center, SRI International, 1983.

[War87] D.H.D. Warren, "OR-Parallel Execution Models of Prolog," *In TAPSOFT'87, The 1987 Int'l Joint Conference on Theory and Practice of Software Development,* Pisa, Italy, 1987.

[Warr88] D.H.D. Warren and S. Haridi, "Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor," *Proc. of the International Conference on Fifth Generation Computer Systems*, 1988.

[WoC86] N. S. Woo and K. M. Choe, " Selecting the Backtrack Literal in the AND/OR Process Model," *Symposium on Logic Programming*, September, 1986.

[Woo85] N. S. Woo, "A Hardware Unification Unit: Design and Analysis," *Proc. of 12th Annual International Symposium on Computer Architecture*, IEEE Computer Society, June 1985.

[YaK84] H.Yasuhara and K.Nitadori," ORBIT: A Parallel Computing Model of Prolog," *New Generation Computing*, vol.2, 1984.

# <부록-1> XWAM-I 인스트럭션에 대한 예상 수행 시간

| Execution Time for Instruction | | | |
|---|---|---|---|
| Instruction | Time | Instruction | Time |
| call | 3R | unify_value_X | 2R+M+2d+u |
| execute | 3R | unify_value_Y | 2R+M+2d+u |
| allocate | R+2M+ts | unify_local_value_X | 2R+M+2d |
| deallocate | R+2M+ts | unify_local_value_Y | 2R+M+2d |
| try_L | 2R+(A+7)M+ts | unify_constant_C | R+M+d |
| retry_L | R+M | unify_nil | R+M+d |
| trust_L | R+2M | unify_integer_V | R+M+d |
| try_me_else_L | 2R+(A+7)M+ts | unify_structure_S | R+2M+d |
| retry_me_else_L | R+M | unify_list_A | R+2M+d |
| trust_me_else_fail | R+2M | unify_end | 2R |
| switch_on_term_L | 2R+d | unify_cdr | R+M |
| get_variable_XA | 2R | cut | R+M |
| get_variable_YA | R+M | set_B | 2R |
| get_value_XA | 4R+u | allocate_cut | R+3M+ts |
| get_value_YA | 3R+M+u | set_cutreg | 2R |
| get_constant_CA | 2R+M+d | try_cut_L | 3R+(A+7)M+ts |
| get_nil_A | 2R+M+d | retry_cut_L | R+2M |
| get_integer_VA | 2R+M+d | trust_cut_L | 2R+2M |
| get_structure_SA | 2R+2M+d | try_me_else_cut_L | 3R+(A+7)M+ts |
| get_list_A | 2R+2M+d | retry_me_else_cut_L | R+2M |
| put_variable_XA | R+M | trust_me_else_cut_L | 2R+2M |
| put_variable_YA | R+M | check_me_else | 2R |
| put_value_XA | 2R | check_independent_XX | 3R+d |
| put_value_YA | R+M | check_independent_XY | 2R+M+d |
| put_unsafe_value_YA | R+3M+d | check_independent_YX | 2R+M+d |
| put_constant_CA | 2R | check_independent_YY | R+2M+d |
| put_nil_A | 2R | check_ground_X | 2R+d |
| put_integer_VA | 2R | check_ground_Y | R+M+d |
| put_structure_SA | 3R+M | start_pcall | R+M(pg+7) |
| put_list_A | 3R+M | spawn_pgoal | 2R+(A+7)M |
| unify_void | R+M | get_pgoal | 3R+(A+14)M+ts |
| unify_variable_X | 2R+M | end_pcall | R+8M |
| unify_variable_Y | 2R+M | back_env_allocate | 9*L*M+3M+2R |

R : register refence
M : memory reference
A : the number of arguments in a goal
d : the time for deference
u : the time for unification
pg : the number of parallel goals in a Par_Frame
L : the number of literals in a clause
ts : the calculation time for determining the position of B or E

# <부록-2> 사용된 벤치마크 프로그램

```
%
% Arch Program
%
arch(V,U,W) :- tower(V), tower(U), tower(W), ne(U,W), on(V,U), on(V,W)

tower(X) :- block(U), on(U,V), top(X,U), bottom(X,V), tower(V), test(X).
tower(X) :- block(X).

test(X) :- ground(X).
test(t(X,Y)) :- test(Y).

ground(d).
ground(f).
ground(g).
ground(i).

on(c,d).
on(a,f).
on(h,i).
on(b,t(c,d)).
on(b,t(a,f)).
on(e,t(h,i)).
on(e,t(a,f)).

top(t(U,V),U).
top(ar(U,V,W),V) :- block(V).

bottom(t(U,V),V).

block(a).
block(b).
block(c).
block(d).
block(e).
block(f).
block(g).
block(h).
block(i).
```

```
%
% Check Program
%
check(X[0]) :- times(X), p(X,Y), p(X,Z), p(X,U), p(X,V), p(X,W), s(Y,Z,U,V,W)
p(X,Y) :- q(X,Y).
p(X,Y) :- r(X,Y).

q(0,a).
q(X,Y) :- X > 0, Z is X - 1, q(Z,Y).

r(0,b).
r(X,Y) :- X > 0, Z is X - 1, r(Z,Y).

s(b,b,b,b,b).
s(a,b,c,b,b).
s(a,a,b,b,b).
s(a,a,a,b,b).
s(a,a,a,a,b).

%
% Data base query
%
query(S,P,C1,C2) :-
        student(S,C1),
        course(C1,D1,R),
        professor(P,C1),
        student(S,C2),
        course(C2,D2,R),
        professor(P,C2), ne(C1,C2).

student(robert,prolog).
student(john,music).
student(john,prolog).
student(john,surf).
student(mary,science).
student(mary,art).
student(mary,physics).

professor(luis,prolog).
professor(luis,surf).
professor(antonio,prolog).
professor(eureka,music).
professor(eureka,art).

professor(eureka,science).
professor(eureka,physics).
```

```
course(prolog,monday,room1).
course(prolog,friday,room1).
course(surf,sunday,beach).
course(maths,tuesday,room1).
course(maths,friday,room2).
course(science,thursday,room1).
course(science,friday,room2).
course(art,tuesday,room1).
course(physics,thursday,room3).
course(physics,saturday,room2).

%
% Fibonacci Program
%
fibo(1,1).
fibo(2,1).
fibo(N,R) :- N1 is N - 1, N2 is N - 2, fibo(N1,R1), fibo(N2,R2), R is R1 + R2.

%
% Eight Queen Program
%
q([A,B,C,D,E,F,G,H]) :-
        eightqueens([p(1,A),p(2,B),p(3,C),p(4,D),p(5,E),p(6,F),p(7,G),p(8,H)])

eightqueens([p(1,A),p(2,B),p(3,C),p(4,D),p(5,E),p(6,F),p(7,G),p(8,H)]):-
    gen(A), gen(B),
    B == A, gen(C), test3(A,B,C), gen(D), test4(A,B,C,D),
    gen(E), test5(A,B,C,D,E), gen(F), test6(A,B,C,D,E,F),
    gen(G), test7(A,B,C,D,E,F,G), gen(H), test8(A,B,C,D,E,F,G,H),
    goodboard([p(1,A),p(2,B),p(3,C),p(4,D),p(5,E),p(6,F),p(7,G),p(8,H)]).

test3(A,B,C):-
    C == A, C == B.
test4(A,B,C,D):-
    D == C, D == B, D == A.
test5(A,B,C,D,E):-
    E == D, E == C, E == B, E == A.
test6(A,B,C,D,E,F):-
    F == E, F == D, F == C, F == B, F == A.
test7(A,B,C,D,E,F,G):-
    G == F, G == E, G == D, G == C, G == B, G == A.
test8(A,B,C,D,E,F,G,H):-
    H == G, H == F, H == E, H == D, H == C, H == B, H == A.

goodboard([]).
goodboard([A|B]) :-goodboard(B),nocollision(A,B).
```

```
nocollision(A,[]).
nocollision(C,[A|B]) :- notoppose(C,A),nocollision(C,B)

notoppose(p(A,C),p(B,D)) :-
        X is C - D, Y is A - B, Z is B - A,
        X == Y, X == Z.

gen(1).
gen(2).
gen(3).
gen(4).
gen(5).
gen(6).
gen(7).
gen(8).

%
% Map Coloring(5 Area)
%
color(A,B,C,D,E) :-
        next(A,B), next(A,C), next(A,D),
        next(B,C), next(C,D), next(B,E),
        next(C,E), next(D,E).

next(yellow,red).
next(yellow,blue).
next(yellow,green).
next(red,yellow).
next(red,blue).
next(red,green).
next(green,blue).
next(green,yellow).
next(green,red).
next(blue,red).
next(blue,yellow).
next(blue,green).

%
% Quick Sort
%
main(Y) :- list(X), qsort(X,Y).
```

```
qsort([], []).
qsort([H|T], S) :-
                        split(H,T,L1,L2),
                        qsort(L2,S2),
                        qsort(L1,S1),
                        append(S2,[H|S1],S).

split(X, [H|Rest], [H|S1], S2) :-
                X < H, !,
                split(X, Rest, S1, S2).
split(X, [H|Rest], S1, [H|S2]) :-
                split(X, Rest, S1, S2).
split(_, [], [], []).

list([27,74,17,33,94,18,46,83,65, 2, 32,53,28,85,99,47,28,82, 6,11,
  11,28,61,74,18,92,40,53,59, 8]).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

---

```
%
% Symbolic Derivation
%
test(Y) :- expression(X), d(X,x,Y).

d(plus(U,V),X,plus(DU,DV)) :- d(U,X,DU), d(V,X,DV).
d(minus(U,V),X,minus(DU,DV)) :- d(U,X,DU), d(V,X,DV).
d(mpy(U,V),X,plus(mpy(DU,V),mpy(U,DV))) :- d(U,X,DU), d(V,X,DV).
d(div(U,V),X,div(minus(mpy(DU,V),mpy(U,DV)),exp(V,2))) :- d(U,X,DU), d(V,X,DV).
d(exp(U,N),X,mpy(mpy(DU,N),exp(U,N1))) :- integer(N), N1 is N - 1, d(U,X,DU).
d(uminus(U),X,uminus(DU)) :- d(U,X,DU).
d(e(U),X,mpy(e(U),DU)) :- d(U,X,DU).
d(log(U),X,div(DU,U)) :- d(U,X,DU).
d(X,X,1).
d(N,X,0) :- ne(N,x).

expression(mpy(3,x)).
expression(plus(exp(x,2),mpy(3,x))).
expression(div(x,2)).
expression(log(x)).
expression(e(x)).
expression(4).
expression(x).
expression(uminus(x)).
```

# <부록-3> 프로세서 사이의 메세지 교환 패턴

o Program : times20
o Run mode : RAP + eager OR
o Total number of PEs used : 6
o Total elapsed time : 12092
o Total number of message sent : 34

| steal | start | redo | fail | kill | success |
|---|---|---|---|---|---|
| 5 | 5 | 9 | 4 | 0 | 11 |

[ 2,1,132,steal ] [ 1,2,176,start ] [ 3,1,186,steal ] [ 1,3,231,start ]
[ 4,1,243,steal ] [ 1,4,286,start ] [ 5,1,297,steal ] [ 1,5,339,start ]
[ 6,1,351,steal ] [ 1,6,390,start ] [ 2,1,3838,success ] [ 3,1,3892,success ]
[ 4,1,3949,success ] [ 5,1,4000,success ] [ 6,1,4051,success ]

**[ 1,2,4170,redo ]* [ 2,1,7568,success ]****
**[ 1,2,7690,redo ]* [ 2,1,7692,fail ]****
**[ 1,3,7699,redo ]* [ 3,1,7710,success ]****
**[ 1,3,7983,redo ]* [ 3,1,7983,fail ]****
**[ 1,4,7992,redo ]* [ 4,1,7995,success ]****
**[ 1,4,8569,redo ]* [ 4,1,8571,fail ]****
**[ 1,5,8578,redo ]* [ 5,1,8583,success ] ****
**[ 1,5,9747,redo ]* [ 5,1,9747,fail ]****
**[ 1,6,9756,redo ]* [ 6,1,9759,success ]****

o Run mode : RAP
o Total number of PEs used : 6
o Total elapsed time : 27626
o Total number of message sent : 38

| steal | start | redo | fail | kill | success |
|---|---|---|---|---|---|
| 5 | 5 | 9 | 4 | 0 | 11 |

[ 2,1,132,steal ] [ 1,2,176,start ] [ 3,1,186,steal ] [ 1,3,231,start ]
[ 4,1,243,steal ] [ 1,4,286,start ] [ 5,1,297,steal ] [ 1,5,339,start ]
[ 6,1,351,steal ] [ 1,6,390,start ] [ 2,1,3838,success ] [ 3,1,3892,success ]
[ 4,1,3949,success ] [ 5,1,4000,success ] [ 6,1,4051,success ]

**[ 1,2,4170,redo ]* [ 2,1,7901,success ]****
**[ 1,2,8023.redo ]* [ 2,1,8109,fail ]****
**[ 1,3,8110,redo ]* [ 3,1,11840,success ]****
**[ 1,3,12114,redo ]* [ 3,1,12201,fail ]****
**[ 1,4,12201,redo ]* [ 4,1,15932,success ]****
**[ 1,4,16501,redo ]* [ 4,1,16587,fail ]****
**[ 1,5,16588,redo ]* [ 5,1,20318,success ]****
**[ 1,5,21480,redo ]* [ 5,1,21567,fail ]****
**[ 1,6,21567,redo ]* [ 6,1,25298,success ]****

# <부록-4> X-WAM 성능 평가 테이블

Program Name : times 10
Query : check
Comment : the first solution, depth = 10

| Number of processors | RAP | | | RAP+IB | | | RAP+EO | | | RAP+EO+IB | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WT | F | I | WT | F | I | WT | F | I | WT | F | I |
| 1 | 25632 | 193 | 2541 | 33562 (25632) | 193 | 2541 | 25632 (25632) | 193 | 2541 | 33562 (25632) | 193 | 2541 |
| 2 | 23576 | 170 | 2803 | 29076 (23576) | 170 | 2803 | 22457 (21530) | 170 | 2131 | 27957 (21530) | 170 | 2131 |
| 3 | 21265 | 136 | 3066 | 26225 (21265) | 135 | 3066 | 18160 (17164) | 136 | 1721 | 23120 (17164) | 136 | 1721 |
| 4 | 18948 | 102 | 3328 | 22368 (18948) | 102 | 3328 | 13881 (12798) | 102 | 1312 | 17301 (12798) | 102 | 1312 |
| 5 | 16634 | 68 | 3591 | 18514 (16634) | 68 | 3591 | 11099 (10022) | 68 | 1407 | 12979 (10022) | 68 | 1407 |
| 6 | 16154 | 35 | 4470 | 16554 (16154) | 35 | 4470 | 8765 (7643) | 35 | 1661 | 9165 (7643) | 35 | 1661 |

I : the number of instructions executed in the processor solving the given query
WT : weighted time
F : the number of failure which occurs in the processor solving the given query
(T) : T is the weighted time without overhead

Overheads and maximum speed up by RAP + EO + IB

| WT | Environment Copy Overhead | Communication Overhead | Backtrack Stack Access + Failure Analysis | Max. Speed up |
|---|---|---|---|---|
| 9165 | 945 (10.3%) | 177 (10.3%) | 400 (4.3%) | 2.91 |

Program Name : times 20
Query : check
Comment : the first solution, depth = 20

| Number of processors | RAP | | | RAP+IB | | | RAP+EO | | | RAP+EO+IB | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WT | F | I | WT | F | I | WT | F | I | WT | F | I |
| 1 | 44392 | 333 | 4441 | 62682 (44392) | 333 | 4441 | 44392 | 333 | 4441 | 62682 (44392) | 333 | 4441 |
| 2 | 40531 | 290 | 4907 | 51767 (40531) | 290 | 4707 | 37678 (36730) | 290 | 3653 | 48914 (36730) | 290 | 3653 |
| 3 | 36166 | 226 | 5373 | 43770 (36166) | 226 | 5373 | 29560 (28564) | 226 | 2861 | 37164 (28564) | 226 | 2861 |
| 4 | 31801 | 162 | 5839 | 39405 (31801) | 162 | 5839 | 21498 (20398) | 162 | 2075 | 29102 (20398) | 162 | 2075 |
| 5 | 27436 | 98 | 6305 | 31408 (27436) | 98 | 6305 | 16641 (15541) | 98 | 2365 | 20613 (15541) | 98 | 2365 |
| 6 | 26660 | 35 | 7972 | 26904 (26660) | 35 | 7972 | 12266 (11144) | 35 | 2828 | 12510 (11144) | 35 | 2828 |

I : the number of instructions executed in the processor solving the given query
WT : weighted time
F : the number of failure which occurs in the processor solving the given query
(T) : T is the weighted time without overhead

Overheads and maximum speed up by RAP + EO + IB

| WT | Environment Copy Overhead | Communication Overhead | Backtrack Stack Access + Failure Analysis | Max. Speed up |
|---|---|---|---|---|
| 12510 | 948 (7.5%) | 174 (1.3%) | 418 (3.3%) | 3.54 |

Program Name : arch program
Query : arch(X, Y, Z)
Comment : the first solution

| Number of processors | RAP | | | RAP+IB | | | RAP+EO | | | RAP+EO+IB | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WT | F | I | WT | F | I | WT | F | I | WT | F | I |
| 1 | 484805 | 7031 | 36865 | 121878 (110753) | 1972 | 7767 | 484805 | 7031 | 36865 | 121878 (110753) | 1972 | 7767 |
| 2 | 473842 | 5905 | 49483 | 105364 (99802) | 846 | 20390 | 509950 (468073) | 5905 | 47536 | 100966 (95404) | 846 | 18510 |
| 3 | 462328 | 4756 | 62106 | 95413 (91705) | 273 | 26173 | 453217 (410392) | 4756 | 44946 | 71788 (68080) | 273 | 17836 |
| 4 | 436804 | 3614 | 76073 | 94159 (91378) | 202 | 27252 | 378205 (334738) | 3614 | 41943 | 70843 (66606) | 202 | 18865 |

I : the number of instructions executed in the processor
solving the given query
WT : weighted time
F : the number of failure which occurs in the processor
solving the given query
(T) : T is the weighted time without overhead

Overheads and maximum speed up by RAP + EO + IB

| WT | Environment Copy Overhead | Communication Overhead | Backtrack Stack Access + Failure Analysis | Max. Speed up |
|---|---|---|---|---|
| 70843 | 1026 (1.4%) | 429 (0.6%) | 2781 (3.9%) | 6.84 |

Program Name : map coloring
Query : map(yellow, X, Y, Z,yellow)
Comment : the first solution

| Number of processors | RAP | | | RAP+IB | | | RAP+EO | | | RAP+EO+IB | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WT | F | I | WT | F | I | WT | F | I | WT | F | I |
| 1 | 6608 | 82 | 382 | 4956 (4616) | 53 | 243 | 6608 | 82 | 382 | 4956 (4116) | 53 | 243 |
| 2 | 5989 | 74 | 372 | 4362 (3688) | 43 | 230 | 6037 (5899) | 74 | 361 | 4430 (3643) | 43 | 215 |
| 3 | 5595 | 64 | 357 | 3970 (3295) | 34 | 206 | 5655 (5466) | 64 | 338 | 4100 (3250) | 34 | 191 |
| 4 | 5453 | 55 | 458 | 3780 (3156) | 25 | 304 | 5543 (5318) | 55 | 443 | 3830 (3114) | 25 | 290 |
| 5 | 5392 | 55 | 452 | 3635 (3030) | 24 | 295 | 5518 (5257) | 55 | 437 | 3740 (2988) | 24 | 281 |
| 6 | 4739 | 41 | 413 | 3350 (2467) | 13 | 263 | 4927 (4604) | 41 | 398 | 3375 (2422) | 13 | 248 |
| 7 | 3574 | 14 | 406 | 2995 (2238) | 10 | 256 | 3811 (3439) | 14 | 416 | 3105 (2193) | 10 | 241 |
| 8 | 3559 | 10 | 555 | 2507 (1945) | 4 | 276 | 3787 (3415) | 10 | 550 | 2750 (1945) | 4 | 271 |
| 9 | 3538 | 7 | 665 | 2328 (1940) | 2 | 365 | 3769 (3397) | 7 | 652 | 2620 (1939) | 2 | 350 |

I : the number of instructions executed in the processor solving the given query
WT : weighted time
F : the number of failure which occurs in the processor solving the given query
(T) : T is the weighted time without overhead

Overheads and maximum speed up by RAP + EO + IB

| WT | Environment Copy Overhead | Communication Overhead | Backtrack Stack Access + Failure Analysis | Max. Speed up |
|---|---|---|---|---|
| 2620 | 239 (9.1%) | 141 (5.3%) | 495 (18.8%) | 2.52 |

Program Name : fibonacci
Query : fibo(10)

| Number of processors | RAP | | | RAP+IB | | | RAP+EO | | | RAP+EO+IB | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WT | F | I | WT | F | I | WT | F | I | WT | F | I |
| 1 | 31794 | 142 | 2400 | 34416 (31794) | 142 | 2400 | 31794 | 142 | 2400 | 34416 (31794) | 142 | 2400 |
| 2 | 19832 | 55 | 34432 | 20865 (19832) | 55 | 3443 | 19895 (19832) | 55 | 3458 | 20928 (19832) | 55 | 3458 |
| 3 | 19833 | 2 | 6506 | 19876 (19833) | 2 | 6506 | 19895 (19833) | 2 | 6515 | 19938 (19833) | 2 | 6515 |
| 4 | 12567 | 2 | 4084 | 12610 (12567) | 2 | 4084 | 12693 (12567) | 2 | 4114 | 12736 (12567) | 2 | 4114 |
| 5 | 12567 | 2 | 4084 | 12610 (12567) | 2 | 4084 | 12693 (12567) | 2 | 4114 | 12736 (12567) | 2 | 4114 |
| 6 | 12567 | 2 | 4084 | 12610 (12567) | 2 | 4084 | 12693 (12567) | 2 | 4114 | 12736 (12567) | 2 | 4114 |
| 7 | 12567 | 2 | 4084 | 12610 (12567) | 2 | 4084 | 12693 (12567) | 2 | 4114 | 12736 (12567) | 2 | 4114 |
| 8 | 8232 | 2 | 2639 | 8275 (8232) | 2 | 2639 | 8421 (8232) | 2 | 2690 | 8464 (8232) | 2 | 2690 |
| 16 | 5664 | 2 | 1783 | 5707 (5664) | 2 | 1783 | 5916 (5664) | 2 | 1855 | 5959 (5664) | 2 | 1855 |

I : the number of instructions executed in the processor solving the given query
WT : weighted time
F : the number of failure which occurs in the processor solving the given query
(T) : T is the weighted time without overhead

Overheads and maximum speed up by RAP + EO + IB

| WT | Environment Copy Overhead | Communication Overhead | Backtrack Stack Access + Failure Analysis | Max. Speed up |
|---|---|---|---|---|
| 5959 | 72 (1.2%) | 180 (3%) | 43 (1.3%) | 5.3 |

# <부록-5> K-Prolog의 구문 구조

```
program  ::= procedure_list
procedure_list ::= prcedure, procedure_List
                 | procedure
procedure ::= clause_list
clause_list ::= clause, clause_list
              | clasue
clause ::= head ':-' literal_list '.'
         | head '.'
head ::= atom
literal_list ::= literal , literal_list
               | literal
literal  ::= atom
           | IDENTIFIER 'is' is_term
           | boolean_literal
           | '!'
atom ::= IDENTIFIER
       | IDENTIFIER '(' term_list ')'
boolean_literal::= add_term rel_op add_term
term_list ::= term '.' term_list
            | term
is_term ::= add_term
term ::= add_term
       | complex_term
add_term ::= add_term add_op mult_term
           | mult_term
mult_term ::= mult_term mult_op simple_term
            | simple_term
simple_term ::= variable
              | INTEGER
variable ::= IDENTIFIER
           | '_'
```

```
complex_term ::= constant
               | skeleton
               | list
skeleton ::= IDENTIFIER '(' term_list ')'
list ::= '[' term list_tail
       | nil
list_tail ::= tail1
            | tail2
tail1 ::= '|' term ']'
tail2 ::= ',' term tail2
        | ']'
nil ::= '[' ']'
constant ::= IDENTIFIER
rel_op ::= '<'
         | '<='
         | '>'
         | '>'
         | '=:='
         | '=\\='
add_op ::= '+'
         | '-'
mult_op ::= '*'
          | '/'
```

# <부록-6> X-WAM 인스트럭션 집합

Procedural Instructions

call P, arity, N
execute P, arity
proceed
allocate
deallocate
halt
cut
fail
nop

Get Instructions

get_variable_X Xi, Aj
get_variable_Y Yi, Aj
get_value_X Xi, Aj
get_value_Y Yi, Aj
get_constant C, Ai
get_nil Ai
get_integer N, Ai
get_structure S, Ai
get_list Ai

Put Instructions

put_variable_X Xi, Aj
put_variable_Y Yi, Aj
put_value_X Xi, Aj
put_value_Y Yi, Aj
put_unsafe_value_Y Yi, Aj
put_constant C, Ai
put_nil Ai
put_integer N, Ai
put_structure S, Ai

put_list Ai

Unify Instructions

unify_void
unify_variable_X Xi
unify_variable_Y Yi
unify_value_X Xi
unify_value_Y Yi
unify_local_value_X Xi
unify_local_value_Y Yi
unify_constant C
unify_nil
unify_integer N
unify_structure S
unify_list
unify_end
unify_cdr

Choice Point Instructions

try_L
retry_L
trust_L
try_me_else_L
retry_me_else_L
trust_me_else fail
switch_on_term_L

AND Parallel Instruction

check me else
check_independent_XX
check_independent_XY
check_independent_YX
check_independent_YY
check_ground_X

check_ground_Y

start_pcall

spawn_pgoal

get_pgoal

end_pgoal

back_env_allocate

Split OR Instruction

entry_point

# <부록-7> X-WAM 인스트럭션의 바이트 코드

| 바이트 코드 | 기호 코드 |
|---|---|
| 1 | call P, arity, N |
| 2 | execute P, arity |
| 3 | proceed |
| 4 | allocate |
| 5 | deallocate |
| 6 | fail |
| 7 | halt |
| 8 | get_variable_X Xi, Aj |
| 9 | get_variable_Y Yi, Aj |
| 10 | get_value_X Xi, Aj |
| 11 | get_value_Y Yi, Aj |
| 12 | get_constant C, Ai |
| 13 | get_nil Ai |
| 14 | get_integer N, Ai |
| 15 | get_structure S, Ai |
| 16 | get_list Ai |
| 17 | put_variable_X Xi, Aj |
| 18 | put_variable_Y Yi, Aj |
| 19 | put_value_X Xi, Aj |
| 20 | put_value_Y Yi, Aj |
| 21 | put_unsafe_value_Y Yi, Aj |
| 22 | put_constant C, Ai |
| 23 | put_nil Ai |
| 24 | put_integer N, Ai |
| 25 | put_structure S, Ai |
| 26 | put_list Ai |
| 27 | unify_void |
| 28 | unify_variable_X Xi |
| 29 | unify_variable_Y Yi |
| 30 | unify_value_X Xi |

```
31          unify_value_Y Yi
32          unify_local_value_X Xi
33          unify_local_value_Y Yi
34          unify_constant C
35          unify_nil
36          unify_integer N
37          unify_structure S
38          unify_list
39          unify_end
40          unify_cdr
41          try L
42          retry L
43          trust L
44          try_me_else L
45          retry_me_else L
46          trust_me_else fail
47          switch_on_term L
48          cut
49          nop
50          entry_point
51          check_me_else_L
52          check_independent_XX Xi, Xj
53          check_independent_XY Xi, Xj
54          check_independent_YX Xi, Xj
55          check_independent_YY Xi, Xj
56          check_ground_X Xi
57          check_ground_Y Yi
58          start_pcall #_pgoals, N, get_addr
59          spawn_pgoal Pid, Arity, Slot
60          get_pgoal
61          end_pgoal
62          back_env_allocate
```

# 여 백

# 제 2 부

# 지식 베이스 개발에 관한 연구

# 여 백

# 제 1 장 서 론

모든 인공 지능 분야에서 지식은 가장 기본 바탕이 되고 있으며, 지식을 표현하고 효율적으로 처리하는 지식 표현의 문제는 인공 지능 분야의 가장 핵심적인 문제가 되고 있다. 그동안 많은 지식 표현 기법이 개발되어 왔는데 [Barr81], 그중 가장 널리 사용되어 온 것이 논리 (logic), 틀 (frame), 의미망 (semantic network), 규칙 (rule)들에 의한 표현 기법들이다.

이와같은 여러가지의 지식 표현 기법들은 각기 나름대로의 장점과 단점을 갖고 있으며 각기 보다 사용하기 좋은 영역을 갖고 있어서, 여러 종류의 지식을 한 시스템 내에 표현하고 그에 바탕을 두고 추론하는 방식에 대한 연구가 활성화 되었는데, 이러한 시도는 크게 두가지 접근 방식으로 분류할 수 있다. 첫번째 방식은 다중 표현 시스템으로 전문가 시스템 개발 도구등에서 많이 사용되는 방식이다. 이 방식은 여러 가지의 표현 방식과 추론 방식을 사용자에게 제공함으로써 사용자가 응용 분야에 맞는 표현 방식을 선택해서 시스템을 구성하도록 하는 방식이다 [Doyl78]. 이러한 복합 방식은 사용자에게 여러가지의 표현 기법을 제공하고 많은 기능을 제공하는 장점이 있으나, 각 표현 형식간의 의미적 결합 관계가 분명히 설정되어 있지 않고 상호 영향등이 고려되지 않는 단점이 있다.

반면에 여러 종류의 지식을 보다 유기적으로 결합하여 표현하고자 하는 방식이 두번째 방식인 혼성 지식 표현 시스템이다. 혼성 지식 표현 시스템은 기본적으로 지식을 구조적으로 기술하거나 정의하여 그 시스템에서 사용하는 용어의 의미를 정확히 할 수 있는 기능과 이렇게 정의된 구조적 지식을 바탕으로 추론하거나 새로운 사실을 증명할 수 있는 기능을 밀접하게 결합하는 방식을 채택하고 있다. 이와 같은 혼성 지식 표현 시스템에는 KRYPTON [Brac83, Brac85a], KL-TWO [McAl82, McAr85], APSN [Kunz84], Cake [Lloy84], BACK [Rich85]등이 있다.

본 연구에서 개발한 Sphinx 시스템 또한 혼성 지식 표현 시스템으로, 대표적인 혼성 지식 표현 시스템인 KRYPTON과 KL-TWO를 결합한 형태를 취하고 있다. Sphinx 시스템 역시 기본적으로 두 개의 구성 요소가 유기적으로 결합된 형태를 취하고 있는데 그중 하나는 구조적인 지식을 표현하기 쉬운 틀과 의미망을 사용한 형식을 제공하며, 포함 관계에 의한 분류에 바탕을 둔 논증 방식을 제공하고 있다.

반면에 다른 하나는 논리 프로그래밍 방식에 의해 구현되었는데, 이는 Horn 논리절로 그 표현력을 제한하는 대신 뛰어난 효율을 얻기 위해서이다. 사실의 표현을 단순히 Horn 논리절로 제한하는 경우에는 그 표현력이 지나치게 제한되기 때문에 Sphinx는 이를 확장한

논리 프로그래밍을 채택하였다. 즉, 부정을 처리함에 있어서 Negation As Failure (NAF) 규칙 [Clan85]을 사용하였으며, 질의에 있어서 일차 술어 논리의 기능을 갖도록, 한정사와 논리적 접속사를 사용할 수 있도록 확장하였다.

또 많은 인공 지능 시스템이 시간에 따라 변하는 지식 베이스를 유지하기 위해 진리 유지 시스템을 갖추고 있는데, Sphinx 시스템 역시 동적인 지식베이스의 변화를 보장하기 위해 진리 유지 방식을 제공하고 있다. 특히 이 시스템에서는 NAF 규칙과 종속 관계에 의한 되돌림에 의한 단순하면서도 효과적인 진리 유지 방식을 개발하여 이를 채택하고 있다. 또한 진리 유지 방식에 필요한 종속 관계를 이용해서 추론 과정에 대한 설명 기능 역시 제공하고 있다.

이외에도 실제로 사용될 수 있는 시스템으로 구성하기 위해 사용자를 위한 여러가지 지식 프로그래밍 환경을 구축하였는데 이에는 지식베이스 편집기와 브라우저등이 있다.

# 제 2 장 Sphinx 시스템

## 2.1 서 론

현재까지의 인공 지능 분야에서 얻어진 가장 큰 결론은 인간의 지적 행위와 활동에 가장 큰 영향을 주고 그러한 지적인 측면을 표상화하는 근본적인 힘은 인간이 갖고 있는 지식에 의한다는 사실이다. 그러나 인공지능의 각 분야마다 필요한 지식의 특성이나 표현 수준이 다르다. 예를 들어, 자연어 처리에서 요구되는 지식은 각 언어의 어휘에 대한 의미를 표현할 지식이나 언어 사용, 사회적 개념, 관습, 개인의 믿음이나 문화적 특성을 나타낼 수 있는 지식의 표현 방식이 요구될 수 있다. 이에 반해 컴퓨터 시각을 위한 지식 표현 방식은 물체의 모양이나 위치, 관계, 재질등을 잘 나타낼 수 있는 기법과 일반적 물리 법칙등을 표현할 수 있는 방법등이 요구될 수 있다. 이와 같이 각 분야마다 필요하거나 유용한 지식 표현 방식이 다름에 따라 그동안 많은 지식 표현 기법이 개발되어 왔는데 [Barr81], 그 중 가장 널리 사용되어 온 것이 논리 (logic), 틀 (frame), 의미망 (semantic network), 규칙 (rule)들에 의한 표현 기법들이다.

이와같은 여러가지의 지식 표현 기법들은 각기 나름대로의 장점과 단점을 갖고 있으며 이에 따라 보다 사용하기 좋은 영역을 갖고 있다. 따라서 1980년대에 들어오면서 인공 지능의 분야가 확장되고 보다 광범위해짐에 따라, 여러 종류의 지식을 한 시스템내에 표현하고 그에 바탕을 두고 추론하는 방식에 대한 연구가 활성화 되었다. 이러한 시도는 크게 두가지 접근 방식으로 분류할 수 있다. 첫번째 방식은 다중 표현 시스템으로 전문가 시스템 개발 도구등에서 많이 사용되는 방식이다. 이 방식은 여러 가지의 표현 방식과 추론 방식을 사용자에게 제공함으로써 사용자가 응용 분야에 맞는 표현 방식을 선택해서 시스템을 구성하도록 하는 방식이다 [Kunz84]. 이 경우에는 주로 '틀' 기법이나 '논리' 기법과 '규칙' 기법을 같이 쓸 수 있도록 하여 단순 규칙에 의한 전문 지식의 표현의 한계를 극복하도록 하고 있다. 그러나 이러한 복합 방식은 사용자에게 여러가지의 표현 기법을 제공하고 많은 기능을 제공하는 장점이 있으나, 각 표현 방식간의 의미적 관계나 표현된 지식간의 의미 관계, 상호 연결, 전체 지식 베이스의 무결성등이 모호하다. 이는 각 표현 형식간의 의미적 결합 관계가 분명히 설정되어 있지 않고 상호 영향등이 고려되어 있지 않기 때문이다. 다중 표현 방식의 지식 프로그래밍 시스템에는 KEE, ART, Knowledge Craft, LOOPS등이 있으며, 이러한 시스템들은 또한 객체 지향, 접근 지향, 규칙 지향, 절차 지향 프로그래밍 파라다임을 결집하여 사용자에게 보다 유익한 기능을 제공하고 있다 [Cutt86].

여러 종류의 지식을 보다 유기적으로 결합하여 표현하고자 하는 방식이 두번째 방식인 혼성 지식 표현 시스템이다. 혼성 지식 표현 시스템은 기본적으로 지식을 구조적으로 기술하거나 정의하여 그 시스템에서 사용하는 용어의 의미를 정확히 할 수 있는 기능과 이렇게 정의된 구조적 지식을 바탕으로 추론하거나 새로운 사실을 증명할 수 있는 기능을 밀접하게 결합하는 방식을 채택하고 있다. 이와같은 방식은 시스템마다 차이가 있으나, 근본적으로 두가지 이상의 구성 요소로 이루어진다는 특징은 같다. 다시 말하면, 한 부분에서는 각 용어 (개념)의 의미를 그 구성 성분이나 다른 용어 (개념)와의 관계에 따라 정의하거나 기술할 수 있는 기능을 갖고 있다. 이에는 주로 틀과 의미망을 기초로 한 표현 방식이 사용된다. 다른 부분은 주로 단정적 기능을 위한 것으로 추론을 할 수 있는 일차 술어 논리에 입각한 정리 증명기가 사용된다. 이 때 혼성 지식 표현 방식이 기존의 지식 표현 시스템과 다른 점은 두번째 부분인 정리 증명기에서, 증명 과정중에 앞에서 말한 정의나 기술에 의한 개념의 의미 구조를 이용한다는 점이다. 그러므로 일차 논리에서는 비효율적인 여러 추론 기능을 보다 단순하고 효율적인 추론 방식으로 대치할 수 있다. 또한 개념의 정의가 추론에 반영되고 또는 새로운 단정적 지식의 획득이 개념의 정의에 영향을 주기 때문에 전체적으로 항상 올바른 의미를 갖고 있을 수 있다. 이와 같은 혼성 지식 표현 시스템에는 KRYPTON [Brac83, Brac85a], KL-TWO [Vila84, Vila85], APSN [McAr85], Cake [Rich85], BACK [Nebe]등이 있다.

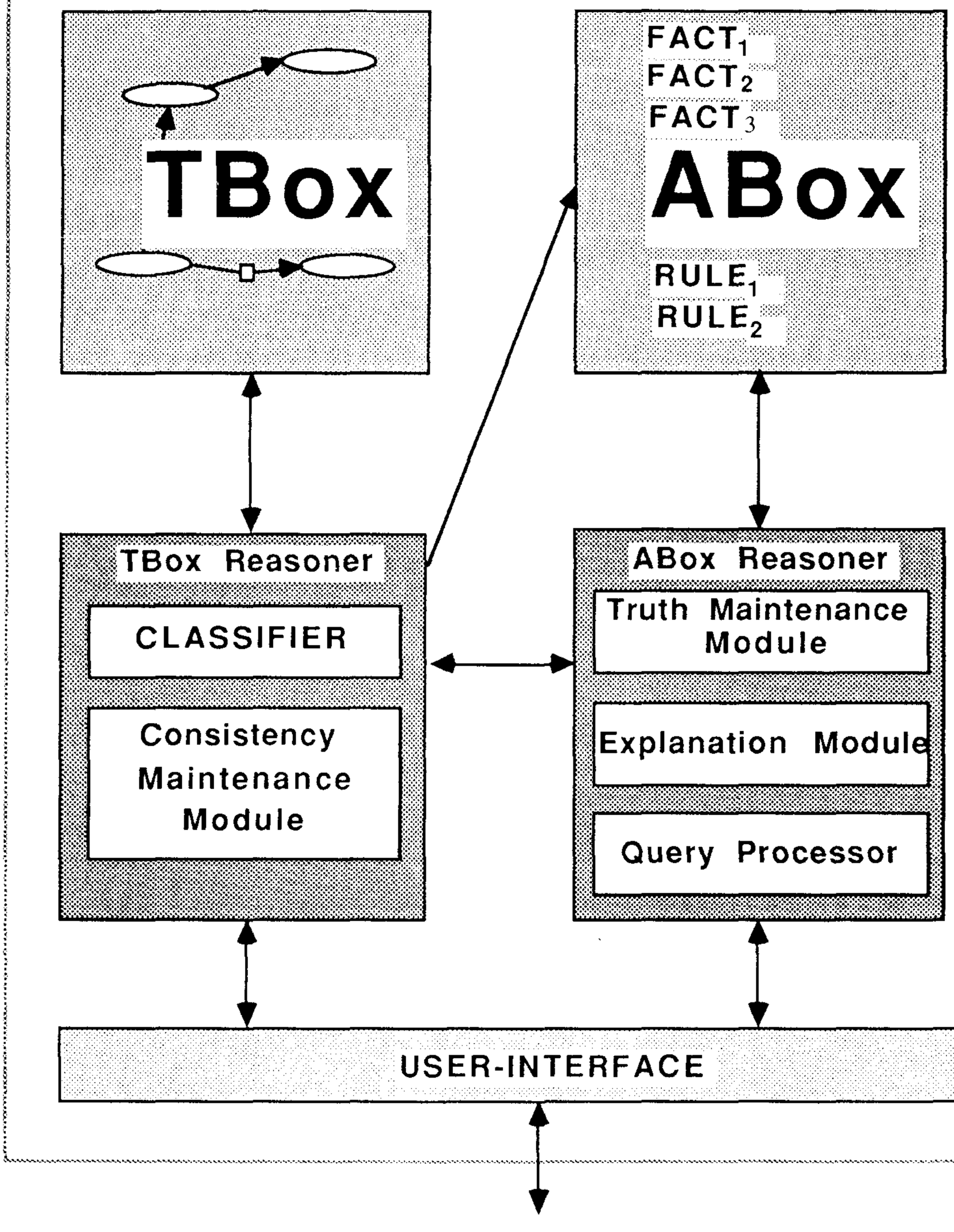
현재까지 알려진 혼성 지식 표현 시스템 중 가장 대표적이라고 할 수 있는 것들은 KRYPTON과 KL-TWO로 이들은 각기 서로 다른 특성을 보유하고 있다. 즉, KRYPTON은 일차 술어 논리 증명기에서 출발하여, 이에 특수한 목적의 추론 기능을 결합한 것이고, KL-TWO는 제한된 기능을 갖는 정리 증명기나 진리 유지기를 바탕으로 여기에 특수 목적의 추론 시스템을 결합하여 표현력은 제한되고 효율은 증대한 시스템을 구축하는 형식을 취하고 있다. KRYPTON에서는 구조적 지식 표현 기법으로 구조적 의미망과 틀을 기본으로 하는 TBox와 단정적 지식을 표현 추론할 수 있는 ABox 두 부분으로 구성되어 있는데, 이 때 ABox에서는 Stickel이 제시한 연결 그래프 정리 증명기 [Stic83]를 사용하였다. 구조적 지식은 ABox에서 정리 증명을 하는 과정, 특히 단일화 (unification) 과정에서 사용하며, 이런 특징에 의해 KRYPTON이 정리 증명기에 바탕을 두었다고 하는 것이다. 다시 말해, 정리 증명기의 기능을 구조적 지식을 표현한 TBox의 기능과 연결하여 보강한 것이다.

KL-TWO는 KRYPTON과 같이 KL-ONE [Brac85b]에서 파생된 시스템이다. BBN에서는 KL-ONE의 의미를 보다 간결하고 명확하게 해서 KRYPTON과 같은 시스템을 구축하기 위해 NIKL이라는 시스템을 구현하였다. KL-TWO는 여기에 MIT에서 개발한

RUP [McAl82]이라는 진리 유지 기능을 갖는 시스템을 결합하여 KRYPTON과 같은 시스템을 구축하였다. 그러나 KRYPTON과 달리 RUP은 일차 정리 증명기가 아닌 명제 논리에 바탕을 두어 그 추론력은 떨어지나 계산적으로는 훨씬 효율적이다. 즉 어느 정도의 표현력을 제한함으로써, 효율을 얻고자 하는 시도인 것이다.

KRYPTON과 KL-TWO와 같은 혼성 지식 표현 시스템의 가장 큰 특징은 두 부분 구성 요소의 연결 방식에 있다. 즉, KRYPTON에서는 추론을 하는 과정에서, 논리 문장으로 표현은 안되어 있지만 구조적으로 표현된 지식을 이용해 단일화가 이루어질 수 있도록 한다. 이에 반해 KL-TWO에서는 하나의 사실이 입력되면, 이 사실에서 유추할 수 있는 가능한 다른 사실들을 어느 정도 추론해서 진리 유지 시스템에 기록한다. 이 때, 구조적 표현 시스템에서 분류 (classification)라는 기본 추론 기법 [Schm83]을 사용해, 현재 언급된 객체를 정확한 위치에 놓도록 한다. 물론 KRYPTON에서도 분류를 사용하지만, 그것은 TBox내에서 요구되는 경우에 한해서 수행되며, 새로운 사실의 입력에 의해 유발되는 것은 아니다. 이것은 KRYPTON에서 점진적인 사실의 첨가가 불가능하기 때문이다. 이때 분류를 특별한 추론 형식으로 다루는 것은 이 추론이 기존의 논리에 의해서도 가능하지만, 그보다는 그래프와 기술에 의한 보다 간단한 방식으로 추론이 가능하기 때문이다 [Schm83].

본 연구에서 개발한 Sphinx 시스템 또한 혼성 지식 표현 시스템으로, 앞에서 언급한 두가지 방식을 결합한 형태를 취하고 있다. Sphinx 시스템 역시 기본적으로 두 개의 구성 요소를 가지며, KRYPTON과 마찬가지로 이들을 각각 TBox와 ABox라고 부른다 <그림 2.1>. TBox는 다른 시스템과 마찬가지로 구조적인 지식을 표현하기 쉬운 틀과 의미망을 사용한 형식을 제공하며, Sphinx에서도 TBox의 기본 논증 방식은 포함 관계에 의한 분류에 바탕을 두고 있다.


TBox
FACT1
FACT2
FACT3
ABox
RULE1
RULE2
TBox Reasoner
CLASSIFIER
Consistency Maintenance Module
ABox Reasoner
Truth Maintenance Module
Explanation Module
Query Processor
USER-INTERFACE
User

<그림 2.1>

반면에 ABox는 논리 프로그래밍 방식에 의해 구현되었는데, 이는 Horn 논리절로 그 표현력을 제한하는 대신 뛰어난 효율을 얻기 위해서이다. 또한 단순 정리 증명기가 아닌 질의 응답 시스템으로서의 역할도 가능하다. 사실의 표현을 단순히 Horn 논리절로 제한하는 경우에는 그 표현력이 지나치게 제한되기 때문에 Sphinx는 이를 확장한 논리 프로그래밍을 채택하였다. 즉, 부정을 처리함에 있어서 Negation As Failure (NAF) 규칙 [Clar78]을 사용하였으며, 질의에 있어서 일차 술어 논리의 기능을 갖도록, 한정사와 논리적 접속사를 사용할 수 있도록 확장하였다. 이는 Lloyd와 Topor에 의한 방식 [Lloy84]과 유사하지만, 질의를 확장한 점과 구조적 지식과 결합한 점이 다르다.

많은 인공 지능 시스템에는 시간에 따라 변하는 지식 베이스를 유지하기 위해 진리 유지 시스템을 갖추고 있다. Sphinx 시스템 역시 동적인 지식베이스의 변화를 보장하기 위해 진리 유지 방식을 제공하고 있다. 특히 이 시스템에서는 NAF 규칙과 종속 관계에 의한 되돌림에 의한 단순하면서도 효과적인 진리 유지 방식을 개발하여 이를 채택하고 있다. 이는 특히 다른 시스템들이 진리 유지를 할 때 구조적 지식을 충분히 사용하지 못하는 결점을 보완해서, 이를 진리 유지에 있어서 충분히 활용하도록 하고 있다. 또한 진리 유지 방식에 필요한 종속 관계를 이용해서 추론 과정에 대한 설명 기능 역시 제공하고 있다.

## 2.2 지식 수준 연산들

지식 수준의 연산은 일반적인 인공 지능 시스템에서 표현하는 지식을 정의하고 검색하고 삭제하는 기본 연산을 말한다. 특히 이는 구현에 관계 없이 그 의미가 명백해야 한다. 따라서 이러한 지식 수준의 연산이 어떤 것들이 제공되는가 하는 것이 인공 지능 시스템의 특성을 분석하는 기본 요소가 된다.

### 2.2.1 TBox의 지식 수준 연산

Sphinx에서의 지식 수준의 연산은 TBox의 연산과 ABox의 연산으로 구분된다. TBox의 연산은 기본적으로 구조 지식의 원소를 정의하는 연산과 이를 바탕으로 정의되는 복합 지식의 정의를 위한 연산, 그리고 이들 간의 관계와 구조에 대한 질의 등으로 구성된다.

TBox에서 구조적 지식을 표현하는 기본 요소로는 개념 (Concept)과 역할 (Role)이

있는데, 개념은 현재 모형화하고자 하는 세계의 객체나 개념, 관계등에 대한 표현이며, 역할은 이러한 개념 사이의 관계에 대한 표현이다. 따라서 개념은 1원 술어와 같으며 역할은 2원 술어와 대응할 수 있다. <그림 2.2>에 원소적 요소의 정의를 위한 지식 수준의 연산이 나타나 있다.

(a) defConcept(C).
(b) defRole(Role, Domain, Range).
(c) defConcept(C1,C2).

<그림 2.2> 원소적 요소의 정의

원소적 요소는 정의할 수 없고 단순히 선언만이 가능한 지식을 표현하는 요소로, 이들은 주로 자연 객체를 표현하는데 사용되는데, <그림 2.2>에서 (a)와 (b)는 가장 간단한 형태의 개념과 역할을 정의하고 있다. (a)는 하나의 원시 개념을 정의하는 것이며 (b)의 경우에는 2원 관계이기 때문에 정의역과 치역을 같이 선언하게 되는데 이들은 모두 개념들이 된다. 예를 들어, child라는 역할은 정의역과 치역을 모두 person이라는 개념에서 찾을 수 있다. 즉 다음과 같이 정의할 수 있다.

defRole(child,person,person)

(c)는 역시 원시 개념인 C2의 하위 개념으로 C1을 정의하는 것으로 C1도 역시 원시 개념이 된다.

이와 달리 복합 지식은 이러한 원소 지식에 의해 정의되는데, 현재 Sphinx에서 정의되어 있는 연산들은 <그림 2.3>과 같다.

(a) defConcpt(C,[C1,...,Cn]).
(b) defConcpt([C1,...,Cn],C), where n >= 2.
(c) defConcpt(C,C1,R,C2).
(d) defRole(R1,R2,Domain,Range).
(e) defRole(R,[R1,R2]).
(f) defConcept(C,C1,[R,1]).
(g) defConcept(C,C1,[R,1],C2).

<그림 2.3> 복합 지식의 정의를 위한 지식 수준 연산

<그림 2.3>에서 첫째 연산은 주어진 리스트에 나타난 개념들의 공통성을 갖는 개념의 정의를 위한 연산이다. 즉, 다음과 같은 의미를 갖는다.

x. (C(x) <-> C1(x) ... Cn(x))

이때, 리스트에 나타난 개념들을 기본 개념 (Basic Concept)이라 하며, 이들은 새로 정의하는 개념인 C의 상위 개념 (super concept) 들이 된다. 예를 들어, '아버지'라는 개념은 다음과 같이 '부모'와 '남성'이라는 개념의 공통성을 갖는다라고 정의할수 있다.

defConcept(father,[parent,male]).

두번째 연산은 개념 C가 C1, ..., Cn로 분할됨을 의미하는데, 따라서 C1에서 Cn까지의 개념들은 상호 배타적이고 오직 이 개념들만이 C의 하위 개념 (sub concept) 들을 형성한다. 이 연산의 의미는 다음과 같이 나타낼 수 있다.

x. (Ci(x) → C(x)) and x. (Ci(x) → ~Cj(x))
where 2 =< i,j <= n and i =/= j

이와 같은 연산은 새로운 지식을 획득하는 경우 오류를 검사하는데 매우 유용하다. 예를 들어, '성'이라는 개념은 '남성','여성','중성'의 세 개념으로 분할될 수 있다.

defConcept([male,female,neutral],gender).

세번째 연산은 개념의 역할 값을 제한함으로써 개념을 특수화하는 것으로, 한 개념을 보다 일반적인 개념을 통해 정의하는 방식이다. 이 연산의 의미는 다음과 같다.

x. (C(x) <-> C1(x) y.(R(x,y) → C2(y)))

예를 들어, '남자'는 그 '성'이 '남성'인 '사람'으로 정의할 수 있다.

defConcept(man,person,gender,male).

이 경우 개념을 특수화 하는데 사용하는 역할을 단순 역할과 구분하여 정의 역할 (definitional role)이라고 부른다. 이 정의 역할은 기본 개념과 함께 정의된 개념의 필요 충분 조건을 형성한다. 따라서 분류 추론에 있어서 기본 관계인, 포함 관계를 표현하는 경우와 지식의 부정시 요구되는 종속 관계를 찾는데 아주 중요한 역할을 한다.

네번째와 다섯번째의 연산은 복합역할의 정의를 위한 연산으로 네번째 연산은 역할 특수화를 나타낸다. 이는 R1이 R2의 특수화된 역할이며, 정의역과 치역이 제한됨을 의미한다. 예를 들어, '아들'이라는 역할은 '자식'이면서 '남성'인 경우를 나타낸다.

defRole(son,offspring,person,male).

다섯번째 연산은 연쇄된 역할을 정의하는 연산으로, 역할의 관계적 합성을 나타낸다. 아래에 그 의미가 나타나있다.

x.y. (R(x,y) <-> z. (R1(x,y)  R2(z,y)))

예를 들어, '손자'라는 역할은 '자식'이라는 두 역할의 연쇄로 정의할 수 있다.

defRole(grandchild,[child,child]).

반면에 6번째와 7번째는 수의 제한을 나타낸다. 즉 6번째는 R인 역할이 반드시 하나 이상 존재해야 하는 C1의 하위 개념인 C를 정의한 것이며, 7번째는 치역이 C2인 R이 반드시 하나 이상 존재해야 하는 C1의 하위 개념 C를 정의한 것이다. 즉 이들의 의미는 다음과 같이 나타낼 수 있다.

x. (C(x) <-> C1(x)  y R(X,Y))

x. (C(x) <-> C1(x)  y (R(x,y) → C2(y)))

예를 들어 '부모'는 자식이 하나 이상 있는 사람으로 다음과 같이 정의할 수 있다.

defConcept(parent,person,[child,1]).

지금까지 살펴본 연산은 TBox에서 지식의 기본 요소인 개념과 역할의 정의에 대한 지식 수준의 연산이다. TBox에는 이와 같이 지식의 정의를 위한 연산 뿐만 아니라 이와 같이 표현된 지식으로 구성된 계층적 구조에 대한, 또는 표현된 구조적 지식들 간의 관계에 대한 질의 역시 필요하다. 이들은 주로 두 개념 사이의 포함관계, 기술에 의한 개념의 검색, 그리고 개념간의 관계를 묻는 연산들로, Sphinx는 이를 위해 <그림 2.4>와 같은 연산들을 제공하고 있다.

(a) subsume(C1,C2).
(b) findConcept(X,[C1,...,Cm],[R1:RC1,...,Rn:RCn]).
(c) find_MSG(C,MSG).
(d) find_MGS(C,MGS).
(e) disjoint(T1,T2).

<그림 2.4> 지식 베이스 검색을 위한 지식 수준 연산들

여기에서 개념간의 포함 관계는 다음과 같이 정의된다.

정의 1) 개념 C1이 개념 C2를 포함하는 경우는 C2의 모든 실체가 개념 C1의 실체가 되는 것을 말한다. 즉,
x. (C2(x) → C1(x))
의 관계가 성립됨을 말하며, $C2 \subseteq C1$ 으로 표시된다.

TBox의 지식 베이스는 개념적으로, 이와 같은 포함 관계에 의해 구성된 구조적 지식의 계층 구조로 되어 있다고 할 수 있다. 이를 바탕으로 어떤 개념이 새로 추가되었을때, 그 위치를 찾고 또는 주어진 기술에 해당 하는 개념이나 역할을 찾는데 사용되는 추론이 분류이며, TBox의 기본 추론 기능이 된다. 이와 같은 분류에 기본한 추론 기능은 새로운 개념을 적절히 위치 시키는 것 뿐만 아니라 새로 정의된 개념이 중복된 것인가를 찾는데에도 역시 유용하다.

<그림 2.4>에서 두번째 질의는 이와 같은 분류 추론을 이용한 가장 일반화된 질의를 나타낸다. 즉, 두번째 인수로 주어진 개념들을 상위 개념으로하고, 세번째 인수로 표현된

역할과 그 한정치를 갖는 개념을 검색하는 질의가 된다.

세번째 질의와 네번째 질의는 주어진 구조에서 첫째 인수로 주어진 개념의 가장 바로 위의 상위 개념들과 바로 밑의 하위 개념들을 구하는 것으로 Sphinx는 이를 바탕으로 자동 분류 (automatic classification)을 수행하고 있다.

다섯번째 질의는 두 개념이 서로 배타적인 관계에 있음을 물어 보는 것으로 개념간의 배타성은 개념을 분할하여 정의할 때 발생하게 된다.

### 2.2.2 ABox의 지식 수준 연산들

Sphinx 시스템의 ABox는 다른 혼성 지식 표현 시스템과 마찬 가지로 TBox에서 표현된 구조적 지식의 외연 (extension)을 저장하고 이를 바탕으로 추론하는 기능을 수행한다. 이와같은 기능을 갖는 지식 표현 형식으로 논리가 가장 적절하다는 것은 널리 알려진 사실이다. 그러나, 완전한 일차 술어 논리는 그 표현력은 강력하나, 시간이 너무 많이 걸리고 기본적으로 증명 과정에 대한 설명 기능이나, 진리 유지 기능이 결여되어 있다. 우리가 혼성 지식 표현 시스템을 통해서 이와 같은 단정적 기능을 갖는 구성 요소를 통해 얻고자 하는 기능은 다음과 같이 요약할 수 있다.

1. TBox 용어의 외연으로 구성된 논리식에 대해 논증할 수 있어야 한다.
2. 변수를 갖고 한정화된 논리식을 증명할 수 있어야 한다.
3. 주어진 질의를 증명할 수 있을 뿐만 아니라 올바른 해를 응답으로 줄 수 있어야 한다.
4. 진리 유지 기능을 갖고 지식을 임의의 순서로 첨가 삭제할 수 있어야 한다.

Sphinx의 ABox와 기존의 정리 증명기와의 차이점은 기본적으로 술어 기호의 의미에서 비롯된다. 즉, ABox에서의 술어 기호는 TBox에서 정의한 개념과 역할이 사용된다. 다시 말해, 인수의 갯수가 1인 술어 기호는 개념에 대응하고, 인수의 갯수가 2인 경우는 역할에 해당하는 것이다.

ABox 내에 표현되는 지식을 크게 두가지로 구분할 수 있다. 하나는 사용자에 의해 첨가된 명제 사실이고 또 다른 하나는 개념이 어떤 역할의 제한 값으로 사용되는 경우나 역할의 연쇄에 의해 어떤 역할이 정의되는 경우에 생성되는 추론 규칙들이다. <그림 2.5>에 ABox 내에 저장된 지식의 일부를 보이고 있다.

```
person(jack).
offspring(jack, mary).
advisor(skhan, jwcho).
phd_student(skhan).
woman(mary).
professor(X) :- advisor(Y,X), graduate_student(Y).
grandchild(X,Y) :- child(X,Z), child(Z,Y).
```

<그림 2.5>

여기에서 추론 규칙들은 TBox 내의 구조적 정의외의 ABox의 추론 기능을 강화하기 위한 것으로, 지식이 다른 지식에 의해 파생되는 경우에는 이를 추론 규칙으로 처리 하는 것이 보다 효율적이라는 결론에 의한 것이다.

ABox에서의 지식 수준 연산은 ask(Q)는 질의 Q가 현재의 지식 베이스에서 증명될 수 있는 가를 묻는 연산이다. 만일 Q가 증명 가능한 경우에는 올바른 대답 치환을 부가적으로 넘겨 준다. 이에 반해 tell_k(K) 연산은 새로운 지식 K를 지식 베이스에 첨가하는 연산이다. 이때, K가 긍정 사실인 경우에는 그 사실이 첨가되고, K가 부정 사실인 경우에는 그 사실과 그 사실에 의존하는 다른 연관 사실들을 삭제하는 효과가 있다. 아래에, 두 연산의 예들을 나타나 있다.

```
?- ask(man(john)).
     yes
?- ask(woman(X)).
     X = mary
?- ask(all(X, father(X) => parent(X))).
     yes
?- tell_k(person(john)).
?- tell_k(not advisor(john, han)).
```

이와 같은 질의을 처리하기 위한 질의 처리 방식은 메타 인터프리터 방식 [Ster86]으로 구현되었으며, 정당함이 증명되어 있다 [Han88].

## 2.3 진리 유지 방식과 설명

과거의 정리 증명기 (theorem prover)들은 점진적 첨가와 제거의 능력이 없었다. 이는 점진적 첨가와 제거가 비단조적 추론을 일으키기 때문인데 이의 해결을 위해 진리 유지 시스템 [Klee86,Doly78,McAl82]이 고안 되었다. 이러한 진리 유지 시스템이 혼성 지식의 표현에도 사용되고 있는데, 그 대표적인 예가 KL-TWO로 여기서는 RUP이라는 진리 유지 시스템을 사용하고 있다.

그러나 진리 유지 시스템을 그대로 사용할 경우 여러가지 문제점들이 발생하게 되는데, 그 중 가장 큰 것은 중간 노드에 대한 정보를 다 갖고 있어야 하기 때문에 지식 베이스가 급격히 증가 함에 따라 해야 할 일들과 기억 공간의 사용량이 급격히 증가 한다는 것이다. 그런데 실제로 이러한 정보들은 TBox의 구조적 정보를 이용하면 다 얻을 수 있는 것 들이다. 따라서 따로 진리 시스템을 두는 것이 아니라 혼성 지식 시스템에 밀접하게 결합되서 지식 베이스 내에 저장된 정보를 충분히 활용할 수 있는 형태의 진리 유지 방식이 필요하다.

이 같은 관점에서 진리 유지 시스템이 해야 할 일들은 크게 세가지로 볼 수 있다. 첫째가 지식을 임의로 첨할 수 있어야 하고, 둘째로 추론된 결과를 설명할 수 있는 기능이 있어야 하고, 세째로 언제든지 지식을 제거할 수 있는 기능이 있어야 한다.

Sphinx에서는 이와 같은 기능을 따로 진리 유지 시스템을 두지 않고 제공해 주고 있다. 첫째로 ABox에 새로운 사실을 첨가하는 것은 매우 쉽게 이루어 진다. ABox가 NAF 규칙을 채택하고 있고 또 추론된 사실들을 기록하고 있지 않기 때문에 단지 첨가된 사실들만을 전제로 기록하면 된다. 둘째로 ABox 내의 사실을 제거하는 것은 부정된 사실을 첨가할 때 이루어지는데, 이 사실이 전제에 속해 있는 경우에는 단순히 제거하면 되고 이 사실이 추론되어 나온 사실이면 이 추론에 바탕이 되는 전제들을 구하여 이들을 제거하면 되는데, 이를 기초 전제 집합(BSP) 이라고 부른다. BSP를 효율적으로 구하기 위해 TBox에 새로운 링크가 도입되는데 한 개념이 TBox에 정의되면 그것의 기본 개념과 정의 역할이 종속 링크로써 연결된다. 세째로 추론에 대한 정당성은 ABox 내의 사실들과 TBox의 구조를 조사함으로 결정된다. ABox 내의 술어 부호들은 TBox 내에 해당 구조를 갖고 있기 때문에 원자식을 증명하는 기초를 TBox의 구조적 정보로 부터 얻어 낼 수 있다. Sphinx에서는 이 정당성을 이용하여 주어진 질의어에 대한 추론 과정을 단계별로 설명해주는 설명 기능을 제공해 주고 있다 [Han88].

## 2.4 시스템 술어들

Sphinx 시스템에서 제공하는 시스템 술어들은 다음과 같다.

### 2.4.1 Concept의 정의와 관련된 연산들

다음은 Sphinx의 TBOX내에 하나의 Concept이나 Role을 정의하는 연산들이다.

1. **defConcept(C)**

   Concept C를 원시 개념 (primitive concept)으로 정의한다. 착오가 발생하지 않았으면 C가 Sphinx의 한 원시 개념으로 정의되었다는 메시지가 출력된다.

   ex)

   | ?- defConcept(thing).

   "thing" is defined as a primitive concept in Sphinx.

2. **defConcept($C_1$,$C_2$)** (*Primitive Specialization*)

   Concept $C_1$을 Concept $C_2$의 하위 개념으로 정의하며 $C_1$는 원시 개념이 된다. 이때 $C_2$도 원시 개념으로 정의되어 있어야 한다.

   ex)

   | ?- defConcept(pc).

   "pc" is defined as a primitive concept in Sphinx.

   yes

   | ?- defConcept(ibm_pc,pc).

   A new link between [ibm_pc] and [pc] is established.

   "ibm_pc" is defined as a primitive concept in Sphinx.

   yes

3. **defConcept(C,[$C_1$, ..., $C_n$])** (*Conjunction*)

   Concept C를 $C_1$, ..., $C_n$의 하위 concept으로 정의한다. 즉 x. (C(x) ≡ $C_1$(x) .. $C_n$(x))가 성립한다.

   ex)

   | ?- defConcept(mammal).

   "mammal" is defined as a primitive concept in Sphinx.

   yes

   | ?- defConcept(thinker).

"thinker" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(person,[mammal,thinker]).

A new link between [person] and [mammal] is established.

A new link between [person] and [thinker] is established.

"person" is defined as a concept in Sphinx.

4. **defRole(R,DC,RC)**

하나의 단순한 role R을 정의한다. R의 domain concept은 DC이고 range concept은 RC이다. 즉 DC에서 RC로 가는 관계 R을 정의한다. 아래의 예는 animal과 gender 사이에 sex라는 관계를 정의한 것이다.

ex)

| ?- defConcept(animal).

"animal" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(gender).

"gender" is defined as a primitive concept in Sphinx.

yes

| ?- defRole(sex,animal,gender).

"sex" is defined as a role in Sphinx.

5. **defConcept([$C_1$, ..., $C_n$],C), where n >= 2** (*Partition*)

현재 정의되어 있는 concept C를 partition하는 concept $C_1$, ..., $C_n$를 정의한다. 즉 x. $(C_i(x) \rightarrow (C(x) \sim C_j(x)))$ where $1 =< i,j =< n, i \neq j$.

ex)

| ?- defConcept(gender).

"gender" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept([male,female,neutral],gender).

A new link between [male] and [gender] is established.

A new link between [female] and [gender] is established.

A new link between [neutral] and [gender] is established.

"male" and "female" and "neutral" is defined as concepts in Sphinx.

6. **defConcept(C,$C_1$,R,$C_2$)** (*Specialization*)

$C_1$의 하위 concept이면서 role R의 range가 모두 $C_2$에 속하는 concept C를 정의한다. 즉 x. (C(x) ≡ $C_1$(x) y. (R(x,y) → $C_2$(y))). 이때 만약 R의 range concept이 RC였다면 $C_2$는 RC의 하위 concept이어야 한다.

ex)

| ?- defConcept(gender).

"gender" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(person).

"person" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept([male,female,neutral],gender).

A new link between [male] and [gender] is established.

A new link between [female] and [gender] is established.

A new link between [neutral] and [gender] is established.

"male" and "female" and "neutral" is defined as concepts in Sphinx.

yes

| ?- defRole(sex,person,gender).

"sex" is defined as a role in Sphinx.

yes

| ?- defConcept(man,person,sex,male).

A new link between [man] and [person] is established.

"man" is defined as a concept in Sphinx.

7. **defConcept(C,[$C_1$, ..., $C_m$],[$R_1$:$RC_1$, ..., $R_n$:$RC_n$])**

가장 복잡한 concept의 정의로 conjunction과 specialization을 서로 합한 것이다. 앞의 $C_i$의 하위 concept이면서 $R_i$의 domain concept이 $RC_i$인 concept C를 정의한다. 즉 x (C(x) ≡ ($C_1$(x) ... $C_m$(x)) $y_1$.($R_1$(x,$y_1$) → $RC_1$($y_1$)) ... $y_n$.($R_n$(x,$y_n$) → $RC_n$($y_n$))).

ex)

| ?- defConcept(mammal).

"mammal" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(thinker).

"thinker" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(gender).

"gender" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept([male,female],gender).

A new link between [male] and [gender] is established.

A new link between [female] and [gender] is established.

"male" and "female" is defined as concepts in Sphinx.

yes

| ?- defRole(sex,mammal,gender).

"sex" is defined as a role in Sphinx.

yes

| ?- defConcept(man,[mammal,thinker],[sex:male]).

A new link between [man] and [mammal] is established.

A new link between [man] and [thinker] is established.

"man" is defined as a concept in Sphinx.

8. **defRole(R,$R_1$,DC,RC)** (*Role Specialization*)

$R_1$의 하위 role, 즉 좀 더 구체화된 role로서 domain concept이 DC이고 range concept이 RC인 role R을 정의한다. 즉, x. y. (R(x,y) ≡ $R_1$(x,y) Domain(R,DC) Range(R,RC)). 이때 $R_1$의 domain concept이 $DC_1$이고 range concept이 $RC_1$이면 RC는 $RC_1$에, DC는 $DC_1$에 각각 포함되야 한다.

ex)

| ?- defConcept(person).

"person" is defined as a primitive concept in Sphinx.

yes

| ?- defRole(offspring,person,person).

"offspring" is defined as a role in Sphinx.

```
yes
| ?- defConcept([man,woman],person).
     A new link between [man] and [person] is established.
     A new link between [woman] and [person] is established.
     "man" and "woman" is defined as concepts in Sphinx.
yes
| ?- defRole(son,offspring,person,man).
     A new link between [son] and [offspring] is established.
     "son" is defined as a role in Sphinx.
```

9. **defRole(R,[$R_1$,$R_2$])** (*Role Chain*)

$R_1$의 $R_2$인 role R을 정의한다. 즉 x. y. (R(x,y) ≡ z. ($R_1$(x,z) $R_2$(z,y))).

ex)

```
| ?- defConcept(person).
     "person" is defined as a primitive concept in Sphinx.
yes
| ?- defRole(child,person,person).
     "child" is defined as a role in Sphinx.
yes
| ?- defRole(grandchild,[child,child]).
     "grandchild" is defined as a role in Sphinx.
```

10. **defConcept(C,$C_1$,[R,1])**

$C_1$의 하위 개념이면서 role R이 반드시 하나 이상 존재해야 하는 concept C를 정의한다. 즉 x. (C(x) ≡ $C_1$(x) y. R(x,y)).

```
| ?- defConcept(person).
     "person" is defined as a primitive concept in Sphinx.
yes
| ?- defRole(spouse,person,person).
     "spouse" is defined as a role in Sphinx.
yes
| ?- defConcept(married,person,[spouse,1]).
```

A new link between [married] and [person] is established.
"married" is defined as a concept in Sphinx.

yes

11. **defConcept(C,$C_1$,[R,1],$C_2$)**

$C_1$의 하위 개념이면서 role R의 range가 $C_2$에 속하는 R이 적어도 하나 존재해야 하는 concept C를 정의한다. 즉 x. (C(x) ≡ $C_1$(x)  y. (R(x,y) → $C_2$(y))).

```
| ?- defConcept(computer).
     "computer" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(os).
     "os" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(unix,os).
     A new link between [unix] and [os] is established.
     "unix" is defined as a primitive concept in Sphinx.
yes
| ?- defRole(support_os,computer,os).
     "support_os" is defined as a role in Sphinx.
yes
| ?- defConcept(unix_computer,computer,[support_os,1],unix).
     A new link between [unix_computer] and [computer] is established.
     "unix_computer" is defined as a concept in Sphinx.
yes
```

### 2.4.2 TBOX의 내용을 검사하는 연산들

1. **subsume($T_1$,$T_2$)**

$T_i$는 concept이나 role이며 $T_1$이 $T_2$를 subsume하는지를 조사한다.

ex)

```
| ?- defConcept(mammal).
     "mammal" is defined as a primitive concept in Sphinx.
```

yes

| ?- defConcept(thinker).

"thinker" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(person,[mammal,thinker]).

A new link between [person] and [mammal] is established.

A new link between [person] and [thinker] is established.

"person" is defined as a concept in Sphinx.

yes

| ?- subsume(mammal,thinker).

no

| ?- subsume(mammal,person).

yes

2. **findConcept(X,[$C_1$, ..., $C_m$],[$R_1$:$RC_1$, ..., $R_n$:$RC_n$])**

$C_i$의 하위 concept이면서 $R_i$의 range concept이 $RC_i$인 concept들 중 가장 상위에 있는 concept이 X이다.

ex)

| ?- defConcept(mammal).

"mammal" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(thinker).

"thinker" is defined as a primitive concept in Sphinx.

yes

| ?- defConcept(person,[mammal,thinker]).

A new link between [person] and [mammal] is established.

A new link between [person] and [thinker] is established.

"person" is defined as a concept in Sphinx.

yes

| ?- defConcept(gender).

"gender" is defined as a primitive concept in Sphinx.

yes

```
| ?- defConcept(gender,[male,female]).
     "gender" is already defined as a concept in Sphinx.
yes
| ?- defConcept([male,female],gender).
     A new link between [male] and [gender] is established.
     A new link between [female] and [gender] is established
     "male" and "female" is defined as concepts in Sphinx.
yes
| ?- defRole(sex,mammal,gender).
     "sex" is defined as a role in Sphinx.
yes
| ?- defConcept(man,person,sex,male).
     A new link between [man] and [person] is established.
     "man" is defined as a concept in Sphinx.
yes
| ?- findConcept(X,[mammal,thinker],[sex:gender]).
X = person
yes
| ?- findConcept(X,[mammal,thinker],[sex:male]).
X = man
yes
```

3. **disjoint($T_1$,$T_2$)**

$T_1$과 $T_2$가 서로 exclusive-or의 관계가 있는지 조사한다.

ex)

```
| ?- defConcept(gender).
     "gender" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept([male,female,neutral],gender).
     A new link between [male] and [gender] is established.
     A new link between [female] and [gender] is established
     A new link between [neutral] and [gender] is established
```

"male" and "female" and "neutral" is defined as concepts in Sphinx.

yes

| ?- disjoint(male,female).

yes

| ?- disjoint(male,gender).

no

4. **find_super(C,Clist)**

   한 concept C의 모든 상위틀들을 depth-first-up-to-join 방식으로 구한 것이 Clist이다.

5. **find_sub(C,Clist)**

   find_super와는 정 반대로 한 concept C의 모든 하위 틀들의 목록이 Clist이다.

6. **find_MSG(C,MSG)**

   concept C의 MSG를 구한다. MSG는 C의 상위 concept들로 이들 가운데는 subsume 관계가 존재하지 않아야 한다.

7. **find_MGS(C,MGS)**

   concept C의 MGS를 구한다. MGS는 C의 하위 concept들로 이들 가운데는 subsume 관계가 존재하지 않아야 한다.

### 2.4.3 ABOX와 관계된 연산들

1. **tell_k(Fact)**

   tell_k는 KB에 새로운 사실을 집어넣는 연산으로, Fact는 C(I)나 $R(I_1,I_2)$ 둘 중의 한 형태이며, C(I)는 I가 concept C의 한 실예라는 의미이고 $R(I_1,I_2)$는 $I_1$와 $I_2$ 사이에 role R의 관계가 있다는 의미이다.

   ex)

   | ?- defConcept(person).

   "person" is defined as a primitive concept in Sphinx.

   yes

```
| ?- tell_k(person(kim)).
     The fact, "person(kim)", is asserted as a premise.
yes
| ?- tell_k(person(lee)).
     The fact, "person(lee)", is asserted as a premise.
yes
| ?- defRole(offspring,person,person).
     "offspring" is defined as a role in Sphinx.
yes
| ?- tell_k(offspring(kim,lee)).
     The fact, "offspring(kim,lee)", is asserted as a premise.
yes
```

2. **ask(Formula)**

ask는 tell_k와는 정 반대로 KB에 들어있는 사실을 문의하는 연산이다. Formula는 일차술어논리의 wff로 다음과 같은 구문을 갖고 있다.

```
AND → &
OR → or
NOT → not
IMPLIES → =>
x. P → forall(X,P)
x. P → some(X,P)
EQUIVALENCE → <=>
ex)
| ?- defConcept(mammal).
     "mammal" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(thinker).
     "thinker" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(person,[mammal,thinker]).
     A new link between [person] and [mammal] is established.
```

```
        A new link between [person] and [thinker] is established.
        "person" is defined as a concept in Sphinx.
yes
| ?- tell_k(person(kim)).
        The fact, "person(kim)", is asserted as a premise.
yes
| ?- ask(person(X)).
X = kim
| ?- tell_k(mammal(lee)).
        The fact, "mammal(lee)", is asserted as a premise.
yes
| ?- tell_k(thinker(lee)).
        The fact, "thinker(lee)", is asserted as a premise.
yes
| ?- ask(mammal(X)&thinker(X)).
X = lee ;
X = kim ;
        no
```

### 2.4.4 KB의 내용을 보는 연산들

1. **show(C)**
   concept에 관한 모든 정보를 보여주는 연산으로, 한 concept C의 정의와 C에 연결된 role들을 보여준다.

2. **show_def(C)**
   한 concept C의 정의를 보여준다.

3. **show_in_english(C)**
   show와 같으나 나타나는 형태가 영어 형태로 나타난다.

4. **list_concept**

현재까지 정의된 모든 concept들의 목록이 나타난다.

5. **list_prim**

현재까지 정의된 모든 primitive concept들이 나타난다.

6. **list_abox**

현재 ABOX내에 저장되어 있는 사실들을 보여준다.

7. **kb_infor**

현재 정의되어 있는 concept의 수와 role의 수를 보여준다.

ex)

```
| ?- defConcept(mammal).
     "mammal" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(thinker).
     "thinker" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept(person,[mammal,thinker]).
     A new link between [person] and [mammal] is established.
     A new link between [person] and [thinker] is established.
     "person" is defined as a concept in Sphinx.
yes
| ?- defConcept(gender).
     "gender" is defined as a primitive concept in Sphinx.
yes
| ?- defConcept([male,female,neutral],gender).
     A new link between [male] and [gender] is established.
     A new link between [female] and [gender] is established.
     A new link between [neutral] and [gender] is established.
     "male" and "female" and "neutral" is defined as concepts in Sphinx
```

```
yes
| ?- defRole(sex,person,gender).
     "sex" is defined as a role in Sphinx.
yes
| ?- defRole(offspring,person,person).
     "offspring" is defined as a role in Sphinx.
yes
| ?- defConcept(man,person,sex,male).
     A new link between [man] and [person] is established.
     "man" is defined as a concept in Sphinx.
yes
| ?- defConcept(woman,person,sex,female).
     A new link between [woman] and [person] is established
     "woman" is defined as a concept in Sphinx.
yes
| ?- defRole(son,offspring,person,man).
     A new link between [son] and [offspring] is established.
     "son" is defined as a role in Sphinx.
yes
| ?- show(man).
Concept "man" is a specialization of [person]
and its Roles are:
     sex : male
     offspring : person
     son : man
yes
| ?- show_def(man).
person and
     whose sex is male
yes
| ?- show_in_english(woman).
person
```

whose sex is female and offspring is person and son is man

yes

| ?- list_concept.

mammal

thinker

person

gender

male

female

neutral

man

woman

yes

| ?- list_prim.

mammal

thinker

gender

male

female

neutral

yes

| ?- tell_k(man(kim)).

The fact, "man(kim)", is asserted as a premise.

yes

| ?- tell_k(person(lee)).

The fact, "person(lee)", is asserted as a premise.

yes

| ?- list_abox.

man(kim)

person(lee)

male(A) :-

sex(B,A)&man(B).

```
female(A) :-
        sex(B,A)&woman(B).
yes
| ?- kb_infor.
The number of defined concepts is 9
The number of defined roles is 3
```

### 2.4.5 유틸리티 연산들

1. **replace(List,Old_Element,New_Element,New_List)**
   List내의 Old_Element 원소들을 모두 New_Element 원소들로 변경하여 New_List를 형성한다.

2. **subtract(List1,List2,List3)**
   List1의 원소들 가운데 List2에 속한 것들을 모두 제거하여 List3를 형성한다. 즉 List1 - List2 = List3.

3. **remove_duplication(List1,List2)**
   List1의 원소들 가운데 중복된 것을 모두 제거한 것이 List2이다.

4. **reverse_list(List1,List2)**
   List1을 역으로 구성한 것이 List2이다.

5. **union(List1,List2,List3)**
   List1과 List2를 합한 결과가 List3이다. 이때 중복된 원소들은 하나만 들어가게 된다 즉 List1 + List2 = List3.

6. **intersect(List1,List2,List3)**
   List1과 List2에 공통적으로 들어 있는 원소들의 리스트가 List3이다.

7. **findall(Var,Clauses,List)**

   Clauses를 만족하는 모든 Var들의 목록이 List이다.

8. **gensym(Root,Atom)**

   Root로 시작하는 atom들을 차례로 만들어 낸다. Root에는 atom 유형의 값이 와야 하며 $Root_1$, $Root_2$의 순서로 심볼들이 만들어지게 된다.

9. **flatten(List1,List2)**

   List1의 원소들 가운데 list 형태로 되어 있는 것들을 모두 원소 형태로 만든 것이 List2이다. 예를 들어 [[1,2,3],[],[[1],[]]]을 flatten시키면 [1,2,3,1]이 된다.

10. **member(Element,List)**

    Element가 List의 한 원소이다. 이때 backtracking되어 올라오게 되면 다음 해를 구하게 된다.

11. **dmember(Element,List)**

    member와 동일하나 해를 한번만 찾고 backtracking되어 올라오면 무조건 fail된다.

12. **append(List1,List2,List3)**

    List1의 끝에 List2를 붙인 형태가 List3가 된다.

13. **list_length(List,Length)**

    List내의 원소들의 수가 Length가 된다.

## 2.5 결 론

여러 지식을 효과적으로 그리고 유기적으로 결합하여 처리하기 위한 Sphinx 시스템은 확장된 논리 프로그래밍 방법과 구조적 지식 표현 방법을 결합하여, 효율성과 시스템의 기능을 강화한 혼성 지식 표현 시스템이다. 특히 확장된 질의어는 한정사와 접속사를 분명히 사용할 수 있도록 하였으며, 대답의 치환을 통해 질의 응답 시스템의 기능을 할 수 있도록 하였다.

진리 유지 경우에는 NAF 규칙과 종속에 의한 되돌림 (recursion)을 통해 보다 효율적인 방식이 되도록 하였다. 이는 특히 커다란 지식 베이스의 경우 저장 공간의 문제를 해결하였다.

Sphinx 시스템의 응용 범위는 기본적으로 보다 이론적으로 정립하고 표현된 지식의 의미가 분명해야 되는 경우에 이의 사용이 가능하다. 이러한 분야는 첫째로 자연 언어의 처리에서 요구되는데, 왜냐하면 대부분의 혼성 지식 표현 시스템과 마찬가지로 Sphinx 역시 자연 언어에서 나타나는 개념의 표현 특히 복합 개념의 표현과 처리에서 뛰어난 기능을 갖고 있기 때문이다. 다른 하나의 분야는 전문가 시스템의 영역으로, 이 경우에 Sphinx는 적용 영역의 배경 지식과 존재하는 객체들에 대한 표현의 추상화에 좋은 기능을 갖고 잇다. 특히 표현된 지식의 지식 수준의 분석이 가능하다는 점은 휴리스틱 분류 [Clan85]에 의한 전문가 시스템 개발에 유용하다.

# 제 3 장 프로그래밍 환경

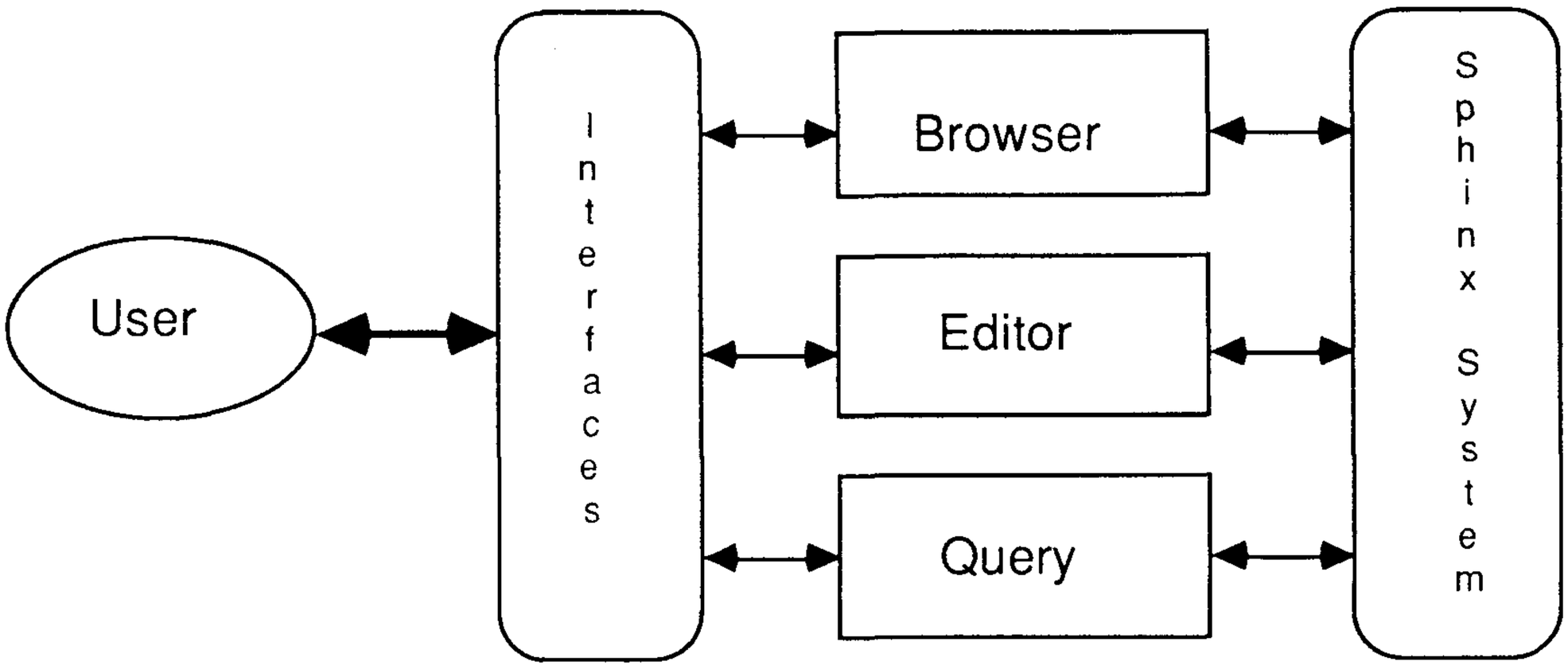
Sphinx의 프로그래밍 환경은 SUN 워크스테이션의 고해상도 스크린과 SunView 윈도우 시스템을 기반으로하여 Quintus Prolog와 ProWINDOWS를 이용하여 다중 윈도우와 메뉴, 그리고 마우스 인터페이스를 제공하고 있다.

Sphinx의 주요 프로그래밍 환경으로는 지식베이스 편집기와 지식베이스 브라우저가 있다. 지식베이스 편집기는 지식 공학자가 Sphinx의 지식베이스를 편리하게 구축할 수 있는 도구를 제공하고, 지식베이스 브라우저는 지식베이스의 구조적 관계를 그래프 형태로 그래픽 스크린에 보여줌으로써 지식 공학자가 지식 베이스의 구조적 관계를 쉽게 이해하고 구성할 수 있도록 해 준다.

## 3.1 Sphinx 시스템 주 인터페이스 (Main Interface)

Sphinx 주 인터페이스는 Sphinx 전체 시스템의 사용자 인터페이스에 해당하며 사용자는 이 Main System을 통하여 Sphinx의 여러가지 기능을 사용할수 있다. 이 Main System은 사용자의 편리를 위하여 윈도우 시스템을 사용하였고 모든 작업은 윈도우 상에서 즉각적으로 이루어진다. 주요 기능은 현재의 모든 지식 처리 시스템이 채용하는 메뉴의 선택( menu-driven) 방식이며, 가능한 그림을 사용하여 표현 함으로써 매뉴얼(Manual) 없이도 사용 가능하도록 하였다.

Sphinx의 전체 시스템의 모습은 <그림 3.1>과 같이 구성되어 있으며 사용자는 이 주 인터페이스를 통하여 브라우저와 지식 편집기 그리고 질의어 처리와 대화하고 또 각 부 시스템들은 Sphinx 주 시스템과 연결되어 있다.




<그림 3.1> Sphinx 시스템과 인터페이스

Sphinx 시스템이 처음 막 시작 되었을 때의 초기 화면은 <그림 3.2>와 같이 Sphinx 시스템의 로고인 Sphinx의 그림과 그리고 현재 디렉토리의 모든 화일을 조사하여 화일의 이름의 끝이 '.kb'인 즉, 지식 베이스 화일의 모든 이름을 아이콘으로 표시하여 로고 그림위에 그려 준다.

초기 화면에서 사용자는 아이콘으로 표시된 지식 베이스 중 하나를 선택함으로써 비로소 시스템이 동작되는데, 만일 지식 베이스가 없는 경우나 또는 새로 만들 경우에는 'creation.kb' 이라고 된 아이콘를 선택함으로 지식 베이스를 새로 구성할수 있다. 지식 베이스가 올라 간후 Sphinx 시스템은 주 화면 상태로 들어가고 기존의 지식 베이스가 올라 간 경우 그 지식 베이스의 구조를 브라우저 화면 상에 그림으로써 작업 준비 상태가 된다. 다음은 지식 베이스가 올라간 후의 주 화면 이다.

Sphinx의 모든 작업은 이 주 화면 상에서 수행되도록 되어 있으며, 주 화면은 크게 네부분으로 구성되어 있는데, 먼저 브라우저 화면, TOOL BOX, 질의란, 그리고 결과 화면으로 되어 있다.

먼저 브라우저 화면에는 지식 베이스의 지식 구조가 그려지는데, 항상 Sphinx Sytem에서 나타나게 되어 있다. TOOL BOX는 Sphinx System이 제공하는 여러가지 기능들을 호출할 수 있는 아이콘들이 그려져 있다. 그러한 아이콘들은 위에서 부터 차례대로 나열하면, 개념 편집기 아이콘, 역할 편집기 아이콘, ABox 편집기 아이콘, Status

아이콘, 지식 베이스 디렉토리 아이콘, 지식 베이스를 저장하는 아이콘, 지식 베이스를 로드하는 아이콘, 그리고 끝으로 Sphinx 전체 시스템을 종료 시키는 Quit 아이콘이 있다.

질의란은 사용자의 여러가지 질의을 입력하는 화면으로 사용자는 이를 통하여, Sphinx 시스템에 질의를 주고 그 추론 결과을 결과 화면을 통해 볼 수 있다. <그림 3.4>는 지식 베이스 디렉토리 아이콘을 누른 후의 화면인데, 여기서 오른쪽 커서 근처에 현재 디렉토리에 있는 지식 베이스의 이름들이 나타나 있는 것을 볼 수있다. 이 중 하나를 선택함으로써 현재 Sphinx 시스템이 가지고 있는 지식을 바꿀 수 있다. 만일 바꾸고자 하는 경우에는 원하는 이름을 선택하면 결과에 'Loading *filename*.....' 이 다 올라 온 후에는 'done' 이 찍힌다. 그리고 이때 브라우저는 바뀌어진 지식에 따라 새로 그려 진다.

다음은 지식 베이스를 저장하는 아이콘을 누른 후의 화면인데, 사용자는 지금까지 Sphinx 시스템에 있는 모든 지식을 저장하고자 하는 경우에 화면 중앙의 Box에 원하는 이름을 만일 Sphinx 외부 형태 (External Representation)로 저장하고자 하는 경우에는 이름의 끝이 '.kb'로 끝나도록 주면 된다. 그외의 경우에는 시스템은 내부 표현 (Internal Representation) 으로 간주하여 저장한다.

지식 베이스를 로드하는 아이콘의 경우에도 Dialogue Box가 나타나면 사용자는 원하는 지식 베이스의 이름을 입력하면 된다. 이때 시스템은 로드될 지식 베이스가 내부 형태인지 외부 형태인지 판단으로 각각에 맞게 자동으로 수행한다.

## 3.2 지식 베이스 편집기 (Knowledge Base Editor)

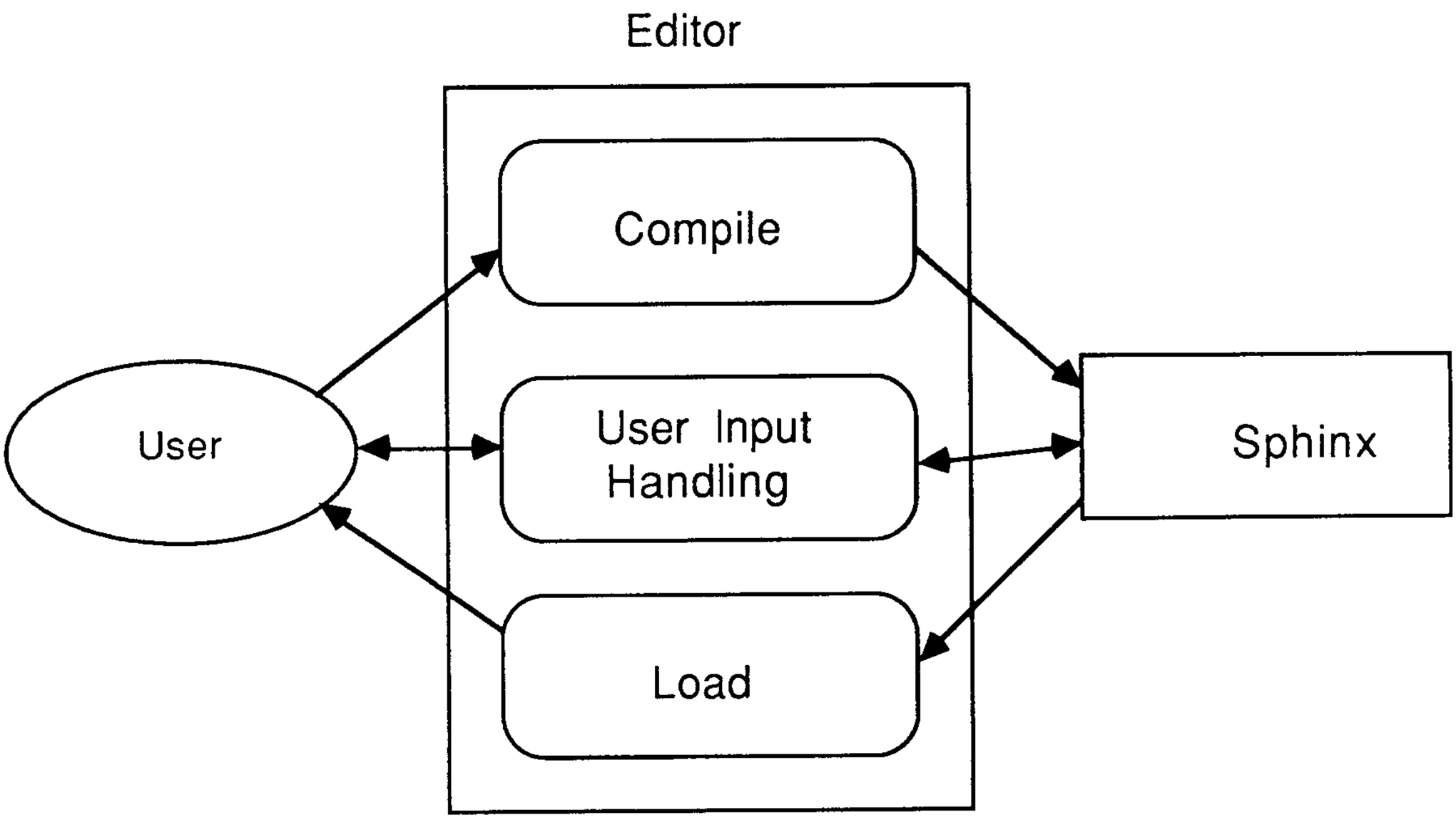
본 지식 베이스 편집기는 사용자에게 최대한의 편의를 제공하는 것을 목표로 구성되었으며, 이를 위하여 프레임에 기초한 편집 방식을 사용하였고 대화식 작업을 가능하게 하였다. 많은 지식 처리 시스템에서는 지식을 표현하는데 있어 사용자에게 편의을 제공하기 위하여, 일반 문서 편집이 아닌 지식 처리만을 주로 하는 시스템을 위하여 서술언어 지향 (Knowledge Base Oriented Language) 편집기를 제공하고 있고, 본 편집기도 Sphinx에서 사용하는 언어에 적합하며 동시에 사용자에게 편리하도록 프레임 구조를 사용하였다. 본 지식 편집기는 지식 처리 시스템에서 필요한 용어적 지식 (Terminalogical Knowledge)과 단정적 지식 (Assertional Knowledge)을 편집하는 도구로써, 편집된 내용은 각 지식 시스템의 내부 표현 (Internal Representation) 형태로 변환되어 후에 추론 시스템에서 사용하게 된다. 이때 새로 추가되는 지식은 기존의 지식들과 일치성을 유지하게 된다.

이 언어 지향 편집기는 단순한 편집 기능외에 컴파일 기능, 지식베이스 일치성 검사,

그리고 편집을 돕기 위해 이미 정의된 개념과 역할들의 리스트를 보여 주는 리스팅 기능 등이 있다. 이러한 기능들은 모두 Sphinx 시스템과 밀접하게 연관되어 총체적인 프로그래밍 환경(Integrated Programming Environment)을 제공하게 된다.

<그림 3.6>은 개념 편집기와 Sphinx 시스템간의 관계를 나타내는 그림이다. 여기서 편집기는 시스템과 사용자의 중간 매개 역할로써 사용자가 프레임 구조에 맞도록 기술해 주면, 그 내용을 다시 Sphinx 시스템으로 보내고 다시 그 결과를 사용자에게 알려 준다. 여기서 편집기는 최대한 시스템의 특수성을 감추고 사용자에게 불편을 주지 않아야 하며, 에러가 발생했을 경우 즉각 알려 주어 사용자의 노력을 줄어야 한다.

지식 편집기는 편집기가 편집을 하는 대상에 따라 세부분으로 구성되어 있는데, 먼저 개념 편집기, 역할 편집기 그리고 ABox 편집기인데, 이는 각각의 편집하는 방법이 매우 다르기 때문이다. 각 편집기에 대해 좀 더 자세히 살펴보면 다음과 같다.




<그림 3.6> Sphinx 시스템과 지식베이스 편집기

### 3.2.1 개념 편집기 (Concept Editor)

개념 편집기는 사용자의 입력을 Sphinx 개념 정의 언어로 바꾸고 또 Sphinx 개념 정의 언어로 된 개념의 기술을 편집기 상에 보여 준다. 편집기는 프레임 구조로 사용자가 정의할 수있는 슬롯을 보여 주는데, 프레임 구조를 나타내는 기술 언어는 다음과 같다.

<Concept> ::= <Concept Name>
{<Sunper Concept>}**n
{<Sub Concept>}*
{<Partition>}*
{<Role name> [NumberRestriction] [ValueRestriction]}**n

사용자는 이런 프레임 구조의 언어를 사용하여 개념을 기술하면 시스템을 이를 Sphinx 시스템의 내부 표현으로 바꾸어 준다. 그리고 사용자가 개념을 수정하거나 보고자 할 경우에는 이의 반대의 변환 과정이 일어난다. 이 기술 언어는 Sphinx의 내부 표현 언어가 프레임 구조에 맞아 떨어지지 않는 문제를 해결하고자 한 것인데, 이를 위하여 프레임 언어에 어떤 항목의 반복을 허용하였고, 내부적으로 이를 저장할수 있는 자료구조인 record를 사용하였다. 여기서 사용하는 자료구조는 각각의 슬롯에 대해 그 내용을 저장하는 record가 딸려 있는데, 이는 나중에 컴파일시에 이용되거나, 개념의 수정시 사용된다. 편집기는 사용자와 Sphinx 시스템의 중간에서 사용자의 입력을 Sphinx 시스템에 돌려 주고 그 결과를 사용자에게 보여주는 역할을 한다.

개념 편집기는 호출은 TOOL BOX의 버튼을 누르거나 혹은 다른 편집기의 개념 편집기 버튼을 선택함으로 가능하다. 이렇게 해서 생성된 개념 편집기는 브라우저의 화면에 나타나게 되고 Quit 버튼으로 사용을 마치면 사라지게 된다. 이 편집기 상에서는 개념의 생성과 수정 그리고 그 결과을 컴파일해 볼 수 있는데, 다음은 생성과 수정 그리고 컴파일을 위한 초기 화면이다.

이 초기 상태에서 사용자는 마우스를 사용하여 개념 편집기 안으로 커서를 이동한후 편집기가 수행가능 상태가 되면 여러가지 연산을 할 수 있다. 다음에서는 개념 편집기의 구조와 버튼의 기능에 대해 언급하고 사용자가 실제로 사용할 기능인 개념의 생성과 수정 그리고 컴파일에 관해 기술하겠다.

### 3.2.1.1 개념 편집기의 구조와 기능

개념 편집기의 슬롯은 전부 6개로 되어 있고, 먼저 Status는 현재 개념 편집기가 편집하는 개념이 새로운 생성된 경우에는 'New'가 그리고 이미 있는 개념이면 'Old'가 표시된다. Concept name은 개념의 이름을 기술하는 슬롯이고, 그다음에는 Super Concept. Sub Concept, Partition 슬롯이 있는데, 이때 Super concept 슬롯에 나타나는 개념은 반드시 이미 정의된 개념이어야 한다. 그리고 Sub concept과 Partition 슬롯의 개념은 반드시 정의되지 않은 새로운 개념이어야 한다. 만일 위의 조건을 어겼을때는 경고 메시지를 보내도록 되어 있다.

개념 편집기에 있는 버튼의 기능은 다음과 같다.

Compile: 개념의 생성 또는 수정이 끝난후 개념을 컴파일 시킨후 지식 베이스에 추가 시킨다.

Abandon: 지금까지의 입력을 지우고 초기 상태가 된다. 이때 입력은 스크린에서 지워지며 무효가 된다.

Quit: 개념 편집기에서 빠져 나올때 사용되는데 편집기는 화면에서 사라진다.

Role: 역할 편집기를 호출한다.

ABox: ABox 편집기를 호출한다.

Load: 개념을 호출할때 사용되며, 이 버튼을 누르면 화면에 Dialogue Box가 나타나게 되는데 사용자는 이때 커서를 마우스를 이용하여 Box 안으로 옮기고 개념의 이름을 입력하면 된다.

LISTING CONCEPT: Sphinx 시스템에 정의된 모든 개념을 리스팅 한다.

LISTING ROLE: Sphinx 시스템에 정의된 모든 역할을 리스팅 한다.

버튼의 이러한 기능은 모든 편집기에서 공통적이며 단지 다른 편집기를 호출하는 것이 편집기마다 다르다. 리스팅 기능은 어느 한 편집기라도 먼저 호출하였으면 그 다음에는 중복을 피하기 위하여 수행되지 않도록 되어 있다.

### 3.2.1.2 개념의 생성

먼저 마우스의 기능을 설명하면 마우스는 세개의 버튼을 가지고 있는데, 먼저 왼쪽 버튼은 캐럿을 움직이게 하는데, 캐럿은 삼각형 모양의 Prompt로 입력의 위치를 표시하며 이곳은 위치가 바뀌면 입력의 위치도 따라 변한다. 가운데 버튼은 아무런 역할이 없고 오른쪽 버튼은 개념 편집기의 PopUp 메뉴를 나타나게 한다. 사용자는 커서를 편집기 위에 놓음으로써 새로운 개념을 생성하기 위한 상태가 끝나게 되며, 사용자는 개념 편집기의 각 슬롯에 따라 기술하면 된다. 먼저 개념의 이름을 입력하고 리턴 키를 치면 캐럿이 자동으로 다음 슬롯으로 넘어가고, Super Concept 슬롯에서는 사용자가 개념의 이름을 치고나면 그 내용을 받아 들이고 다음 내용을 받아들이기 위해 앞전의 내용을 지운다. 만일 더 입력할 내용이 없을 경우에 리턴을 치면 다음 슬롯으로 넘어가게 되어 있는데 이때 가장 마지막에 입력한 내용이 슬롯에 나타난다. Sub Concept 그리고 Partition 역시 위와 비슷하게 동작하고 Role에서는 먼저 Role의 이름을 기술하며 캐럿이 자동으로 Number Restriction 슬롯으로 넘어가는데, 만일 기술할 내용이 없으면 리턴을 치면 된다. 그 다음 커서는 Value Restriction 슬롯으로 넘어가고 그 다음에는 다시 처음의 Role 이름 슬롯으로 넘어가고 다른 Role을 기술할 수 있게 된다. 이때 리턴을 치면 커서는 처음의 Concept Name 슬롯으로 간다. 잘못 입력한 경우는 개념의 수정에서 다음 절에서 다루기로 하겠다.

### 3.2.1.3 개념의 수정

이미 정의된 개념을 수정하기 위해서는 먼저 Sphinx 시스템 상에 기록된 개념을 호출해야 하는데, 여기에는 두가지 방법이 있다. 먼저는 개념 편집기에 있는 Load 버튼을 누르면 아래와 같은 화면이 되고 그 중앙에 Dialogue Box가 나타나는데, 이때 사용자는 호출한 개념의 이름을 입력하면 된다.

또다른 방법으로는 사용자가 처음에 개념의 이름을 지정할 때 만일 이 이름이 이미 정의된 이름이면, 사용자가 그 개념을 호출한 것으로 간주하고 그 개념을 자동적으로 올리도록 되어 있다. 이렇게 호출된 개념에 대해 직접 화면 상에서 수정이 가능한데 <그림 3.9>는 개념 호출을 통해 개념 편집기에 나타난 successful_father의 개념 정의이다.

이 개념의 원래 Sphinx 시스템 내에서의 정의는 다음과 같다.

defConcept(successful_father, father, married_daughter)

이러한 Sphinx 개념 정의가 위와 같이 바뀌어 나타난다. 이렇게 호출된 개념은 먼저 개념의 이름을 고치고자 하는 경우 캐럿을 Concept name 슬롯의 안으로 옮겨 놓은후 Back Space 키를 이용하여 지운후 고쳐 쓸 수 있다. 그 다음에는 Super concept, Sub concept, 그리고 Partition의 경우 사용자의 입력이 지워지고 그 다음 새로 쓰여지기 때문에 이전 입력을 볼수가 없게 되어 있다. 이를 보완하기 위하여 개념 편집기에서 오른쪽 마우스의 버튼을 누르면 PopUP 메뉴가 나타나고 그중 하나를 선택하면 Window가 생성되고 정의된 내용을 보여 준다. <그림 3.10>에서는 커서 다음에 나타나는 Box가 PopUp 메뉴이다.

Window에서 한 슬롯을 선택하면 그 슬롯의 내용이 개념 편집기에 나타나고 그 위에서 수정 가능하다. <그림 3.11>에서는 Partition에 이미 'all_being_is_being'과 'all_nonbeing_is_nonbeing'을 사용자가 입력하였는데, 사용자가 'all_being_is_being'을 'all_being_is_being'로 고치기 위해 먼저 오른쪽 버튼으로 PopUp 메뉴를 부른 후 이 중 Partition를 선택하면, 화면에 Partition 슬롯에 대해 지금까지 입력된 내용이 나타나는데, 이 중 고치려는 하나를 커서를 가지고 선택하여 그 내용인 'all_being_is_being'가 다시 Partition 슬롯에 나타난 것을 보여 주고 있다.

이렇게 만들어진 화면은 다시 PopUP를 보면 Partition_Off 라고 되어 있는데, 이를 다시 선택하면 화면은 없어지고 PopUp 메뉴는 Partition으로 바꾸어 진다. 일단 Window에서 한 항목을 선택한후 그 항목이 개념 편집기의 해당 슬롯에 나타나게 되는데, 이때 그 항목은 지워지게 되어 있다. 그래서 수정이 끝난후 또는 다시 그 내용을 기억 시키고자 할 경우는 다시 그 술롯에 마우스를 사용하여 캐럿을 옮긴후 리턴 키를 쳐야 한다. 사용자의 편리를 위하여 이미 지식 베이스에 기록된 개념과 역할을 보여 주는 리스팅 기능이 있는데, 이를 사용하여서도 입력이 가능하다. 먼저 사용자는 개념 편집기 상의 LISTING CONCEPT나 LISTING ROLE 버튼을 선택하면 그러한 리스트를 보여주는 Window가 생성되어 나타나게 된다. 이를 사용하여 입력하고자 하는 경우 먼저 마우스로 캐럿을 원하는 입력 슬롯에 옮겨 놓은후 개념 리스트나 역할 리스트 상에 나타난 이름들을 마우스로 선택하면 그 항목의 이름이 해당 슬롯에 나타나게 된다. 이때 사용자는 필요에 따라 수정할 수도 있고 그냥 고칠 필요가 없은 경우 리턴을 치면 된다. 다음은 개념 리스팅 기능을 사용하여 입력을 하는 한 예를 보여주는데, 사용자는 'boy'을 정의하면서 그것의 Super Concept의 'male'를 손으로 입력하는 대신에, 개념 리스트에 나타난 'male'를

커서로 선택함으로써 편집기 상에 나타난 것을 보여 주고 있다.

### 3.2.1.4 개념 편집기에서의 컴파일

위에서 입력한 개념을 지식 베이스에 추가하고자 할 경우 개념을 컴파일 시켜야 하는데, 이는 편집기의 컴파일 버튼을 누름으로써 가능한데, 결과는 결과 화면에 'end of compile'이라는 메시지가 나타난다.

### 3.1.1.5개념 편집기에서의 경고 메시지

사용자에게 에러가 생기는 즉시 고칠 수 있도록 편집기는 사용자에게 다음과 같은 경고 메시지를 제공한다.

Error !, Concept Name is Not defined : Concept name을 입력하지 않고 개념을 정의하려고 한 경우

Error !, Super Concept is Not Defined : Super Concept의 입력시 사용자가 입력한 개념이 정의되지 않은 경우이다.

Error !, Sub Concept is Already Defined :Sub Concept 입력시 이미 정의된 개념을 입력한 경우

Error !, Parted Concept is Already Defined : Partition 입력시 이미 정의된 개념을 입력한 경우

Error !, Incorrespondence between Super and Role ; 컴파일시 체크되는 경고인데. 입력된 Super concept의 수가 Role의 수와 일치하지 않을 때

모든 경고 메시지는 화면의 중앙에 Box가 그려지고 그 안에 경고 메시지가 찍혀 있다. 사용자는 경고를 보고 에러를 수정한 후 Box 안의 'Cancel' 버튼을 누르면 사라진다. <그림 3.13>은 Super concept으로 입력한 'boy'가 정의되지 않아 경고 메시지가 나타난

경우이다.

### 3.2.2 역할 편집기 (Role Editor)

역할 편집기 역시 사용자가 프레임에 맞게 기술한 내용을 Sphinx의 내부 표현으로 바꾸어 준다. 이때 프레임의 구조를 기술하는 언어는 다음과 같다.

```
<Role> ::= <Role name>
           [<Domain> <Range>]
           [<Role Composition from> <To>]
```

그리고 역할을 보여줄때는 반대의 변환 과정이 일어난다. 역할 편집기 역시 TOOL BOX의 선택이나 혹은 다른 편집기의 선택에 의해 호출될 수 있다. 여기에서는 역할의 생성과 수정 그리고 컴파일을 수행하는데, 초기 화면은 아래와 같다.

여기서도 만일 사용자가 이미 정의된 역할의 이름을 입력하면 자동적으로 그 역할을 호출하도록 되어있다. 버튼의 기능은 개념 편집기와 같으며 만일 개념 편집기가 호출되지 않았을 경우에 역할 편집기에서도 호출할수 있다. (자세한 버튼의 기능은 앞의 3.2.1을 참고 할것)

역할 편집기는 전부 5개의 슬롯으로 구성되어 있고, Status는 개념 편집기처럼 새로운 개념이면 'New'을, 이미 정의되었던 개념이면 'Old'을 표시한다. Domain과 Range는 반드시 정의된 개념이어야 하며, 그렇지 않은 경우 경고 메시지가 나타난다. Role Composition의 경우는 두 개의 역할이 이미 정의된 역할들 이어야 한다. Domain이 정의되면 반드시 Range도 같이 정의되야 하며, Role Composition의 경우도 둘 다 반드시 정의된 역할이어야 한다.

#### 3.2.2.1 역할의 생성

역할을 생성하기 위해서는 개념 편집기에서와 같이 먼저 마우스의 커서를 역할 편집기 안으로 옮기고, 먼저 Role Name 그리고 다음의 슬롯들을 만일 필요하면 입력하고 필요없는 경우 리턴 키를 치면 다음 항목으로 자동으로 옮겨 간다. 역할 편집기에서의 입력도 개념 편집기에서와 비슷하며 훨씬 단순하다.

#### 3.2.2.2 역할의 수정

역할의 수정을 위해서 먼저 역할을 지식 베이스에서 가져 와야 한다. 가져 오는 방법은 개념 편집기에서와 같이 Load 버튼을 사용할 수도 있고 바로 역할의 이름을 입력하며 자동적으로 역할 편집기 상에 정의된 내용이 나타난다. 이렇게 화면 상에 나타난 역할을 캐럿을 이동하여 슬롯 내에서 Back Slash 키를 이용하여 지우고 다시 쓰면 된다. 다음은 이미 정의된 역할을 역할 편집기로 가져온 예이다.

#### 3.2.2.3 역할의 컴파일

지금까지 정의된 역할을 Sphinx 시스템에 추가하기 위해서는 컴파일을 해야만 한는데, 역할의 컴파일은 역할 편집기의 컴파일 버튼을 누름으로써 이루어지며 이때 결과는 결과 화면에 'end of compile' 이라고 찍힘으로써 컴파일이 성공적으로 끝났음을 나타낸다.

#### 3.2.2.4 역할 편집기에서의 경고 메시지

역할 편집기에서의 경고 메시지는 다음과 같은 경우가 있을 수 있다.

Error !, Role name is Not defined : Role name을 정의하지 않은 경우

Error !, Role Domain is Not defined : Role Domain이 정의되지 않은 경우

Error !, Role Range is Not defined : Role Range가 정의되지 않은 경우

Error !, Super Role name is Not defined : Super Role의 입력된 역할이 아직 정의되지 않은 역할인 경우

Error !, Role From is Not defined : Role Composition에서 From에 해당하는 역할이 아직 정의되지 않은 경우

Error !, Role To is Not defined : Role Composition에서 To에 해당하는 역할이 정의되지 않은 경우

Error !, Role Domain is missed : Role Range만 정의된 경우

Error !, Role Range is missed : Role Domain만 정의된 경우

Error !, Role From is missed : Role Composition에서 To만이 정의된 경우

Error !, Role To is missed : Role Composition에서 From만이 정의된 경우

### 3.2.3 ABox 편집기 (ABox Editor)

ABox 편집기도 역시 Sphinx의 단정적 기술 언어와 사용자에게 편리한 언어 사이에서 매개 역할을 하는데, ABox 편집기의 언어는 단순하다. 먼저 개념이나 역할을 기술하고 다음에 적당한 실제 예 (Instance)를 빈칸이나 '(' 또는 ':' 등의 문자로써 구분해주면 된다. 이 편집기 역시 TOOL BOX나 다른 편집기의 선택에 의해 호출된다. 여기에서는 개념 편집기와 역할 편집기에서 정의한 지식들을 가지고 실제 지식을 표현하는 ABox의 내용을 편집하는데, 여기서는 실제 지식의 입력, 수정, 그리고 추가와 컴파일이 가능하다. 다음은 ABox 편집기의 초기 화면이다.

대부분의 버튼의 기능은 개념 편집기와 유사하다. (개념 편집기 3.1.1.1를 참조)

#### 3.2.3.1 ABox 편집기에서의 생성

여기서는 먼저 개념의 예 (Instance)를 기술할 경우에는 먼저 캐럿을 ABox 편집기 안으로 옮긴후 먼저 개념을 입력한후 space나 '(' 또는 ':'를 친후 예를 입력하면 된다. 역할 예를 기술하는 경우에는 먼저 역할의 이름을 친후 다음 위에서와 같은 delimeter를 친후 Domain의 예 (Instance) 그리고 Range의예 (Instance)를 기술하면 된다. 이때도 개념 편집기나 역할 편집기에서 처럼 리스팅 기능을 사용하여 편리하게 입력 할수 있는데, 먼저 개념과 역할의 리스트를 만든후 항목을 마우스로 선택하면 현재 ABox 편집기의 캐럿이 있는 위치에 나타나게 되어 있다.

#### 3.2.3.2 ABox 편집기에서의 수정

수정은 먼저 'Load' 버튼을 사용하여 지식 베이스의 내용을 올린후 마우스을 사용하여 캐럿을 원하는 위치로 옮긴후 수정할수 있다. 다음은 Load 버튼에 위해 ABox 편집기 상에 나타난 단정적 지식 예이다.

#### 3.2.3.3 ABox 편집기에서의 컴파일

생성되거나 수정된 단정적 지식을 Sphinx 시스템에 추가시키기 위해 컴파일이 필요한데, 컴파일은 편집기의 컴파일 버튼을 누름으로써 이루어 지며 이때 컴파일의 결과는 결과 화면에 'end of compile'를 찍음으로써 올바르게 수행되었음을 나타난다.

#### 3.2.3.4 ABox 편집기에서의 경고 메시지

ABox 편집기에서 가능한 경고 메시지는 아래와 같다.

Error !, No exist Concept name : Concept name이 정의되지 않은 경우

Error !, No exist Role name : Role name이 정의되지 않은 경우

Error !, No argument(s) : 필요한 인수가 부족 할때

<그림 3.18>은 ABox 편집기에서 잘못 입력한 경우 발생한 경고 화면이다.

전부 세개의 편집기가 Sphinx 정의 언어를 편집하고 수행시키는데 보조하며 이들 모두를 한껏번에 화면 상에 호출하여 사용할 수도 있다. <그림 3.19>는 세개의 편집기가 모두 사용되는 것을 보여 주고 있다.

## 3.3 지식 베이스 브라우저

Sphinx와 같은 혼성형 지식 표현 시스템에서는 개념들 사이의 관계를 표현하기 위해서 의미망을 이용한다. 이때, 개념들은 그래프(Graph)의 노드로, 그리고 개념들 사이의 관계들은 링크로 보여질 수 있다. 대부분의 지식 프로그래밍 환경은 이와 같은 의미망 구조를 고해상 그래픽 스크린을 이용하여 프로그래머에게 친근하게 보여주고, 마우스 등의 고급 인터페이스를 사용하여 그래프 상에서 의미망 구조를 임의적으로 변경할 수 있도록 해준다. 이러한 기능을 갖는 도구를 일반적으로 브라우저(browser)라고 한다.

그래픽 브라우저는 인공 지능 시스템에서 표현하는 지식 베이스의 구조 상태나 추론 과정등을 그래픽을 통해 사용자가 보다 이해하기 쉬운 형태로 보여주는 기능을 갖고 있다. 또한 그래픽 브라우저는 보여주는 기능이외에 특정한 항목을 선택하여 그에 대한 여러

기능을 수행할수 있도록 한다. 보다 발전된 브라우저는 그래픽으로 표현된 구조관계를 변형 시킴으로써 내부 구조 관계를 직접 변경할 수 있는 편집 기능도 갖고 있다.

대부분의 인공 지능 시스템은 뛰어난 사용자 인터페이스를 갖고 있기 때문에 이와 같은 그래픽 브라우저를 고해상도의 그래픽 스크린에서 제공하고 있다. Sphinx는 Sun3/160 워크스테이션에서 Quintus Prolog와 ProWINDO를 사용하여 개발하였다.

### 3.3.1 Sphinx 브라우저의 기능

Sphinx는 논리에 기초한 지식 표현 시스템이지만 근본적으로 의미망 개념을 사용하여 개념들과 역할들을 표현하기 때문에 이들의 관계는 형태로 자연스럽게 보여질 수 있다. Sphnix에 지식 베이스 브라우저를 위한 그래픽 기능을 첨가하기 위해 prolog 언어 자체에 윈도우 프로그래밍 기능이 포함되어 있는 ProWINDOWS를 사용한다.

Sphinx 브라우저는 보다 편리한 프로그램 환경을 제공하기 위해서 다음과 같은 기능을 갖는다.

a) 지식의 구조적 관계를 보여주는 기능

Sphinx에서 존재하는 구조적 관계에는 개념들간의 상-하의 계급적 관계와 개념과 개념 사이의 역할 (role) 관계가 있다. 따라서 이를 그래프 형태로 표시하기 위해 다음과 같은 두가지 형태의 네트워크을 제공한다.

1) 개념 네트워크

개념들을 노드로하고 이들 간의 계급적 관계를 링크로 표시한다. 이때 상위 개념은 하위 개념의 왼쪽에 위치하게 된다.

2) 역할 네트워크

역할은 기본 역할과 역할 특수화에 의해 정의된 역할 그리고 역할간의 연쇄에 의해 정의된 역할이 있다. 역할 네트워크는 이러한 역할들을 노드로 표시하고 이들 간의 계층적 관계를 링크로 표시한다.

b) 윈도우 기능

Sphinx에서는 지식 베이스를 사용자에게 보다 편리하게 보여주기 위해서 다음과 같은 윈도우 기능을 갖게 된다.

1) 윈도우의 위치 이동 및 크기 변환

사용자가 지식 베이스의 구조를 참조하면서 동시에 관련된 다른 작업을 하기 위해서는 브라우저 윈도우를 원하는 위치에 원하는 크기로 조정할 수 있어야 한다.

2) 윈도우 내에서 scrolling

지식 베이스의 관계를 나타내는 그래프가 실제로 보여질 윈도우 보다 크면 그래프의 일부만 보여질 수 있다. 이때 그래프의 다른 부분을 볼 수 있도록 상-하 또는 좌-우 방향의 scrolling 기능을 제공한다.

3) 윈도우 아이콘

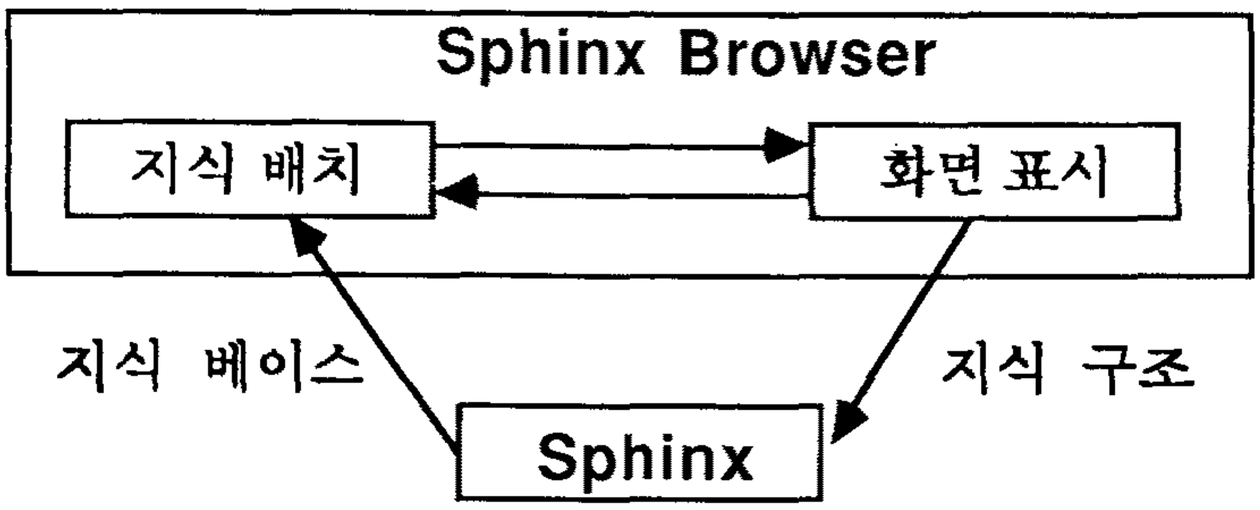
브라우저 작업을 일시적으로 보류하고자 할 때는 윈도우를 아이콘화 하여서 임의의 위치에 옮길 수 있고, 다시 브라우저 작업을 할 때는 아이콘을 열어서 원래의 상태로 회복할 수 있다.

4) 다중 윈도우 기능

네트워크가 클 경우에 여러 윈도우를 이용하여 여러 부분을 동시에 참조하면서 작업을 할 수 있다.

### 3.3.2 Sphinx Browser의 구성

Sphinx 브라우저는 <그림 3.20>처럼 지식 배치 부분과 화면 표시 부분으로 구성된다. 지식 배치(layout) 부분은 Sphinx로 부터 지식 베이스의 구조를 넘겨 받아, 각 개념이 윈도우 상에 표시될 위치를 계산한다. 화면 표시 부분은 배치 부분에서 구해진 위치를 이용하여 개념을 윈도우의 해당 위치에 표시하고 개념 사이의 구조적 관계를 링크를 이용하여 표시한다.




<그림 3.20> Sphinx 브라우저의 구조

**3.3.2.1** 지식 배치

브라우저가 그려줄 대상이되는 지식 베이스나 추론 정보는 다음과 같은 형태로 Sphinx로 부터 받게된다.

a. [isa, Concept1, Concept2]
b. [role, Role, Domain, Range]
c. [roleisa, Role1, Role2]
d. [exist_role, Role, Concept1, Concept2]
e. [exist_role. Role, Concept]
f. [concept, Concept]
g. [role_chain, Role, Role1, Role2]

Sphinx는 위와 같은 지식망을 개념의 내포 관계에 따른 순서쌍의 집합으로 바꿔준후에 배치 알고리즘을 수행하게된다. Sphinx에서 사용하는 배치 알고리즘은 USC/ISI에있는 Gabriel Robins의 ISI Grapher 알고리즘을 토대로하여 개발되었다. ISI Grapher 알고리즘은 격자(lattice) 구조의 그래프를 2차원 평면 상에 디스플레이 하기위해 개발된 소프트웨어 도구이다. 그리고 로빈스는 임의의 방향성을 갖는 그래프를 선형 시간(linear-time complexity)에 좌표를 구할수 있음을 증명하였고 널리 응용 분야들을 통해서 질적으로 보기에 좋은 디스플레이를 한다고 인정을 받았다 [Robi88]. 다음은 ISI Grapher 알고리즘을 개략적으로 기술하였다.

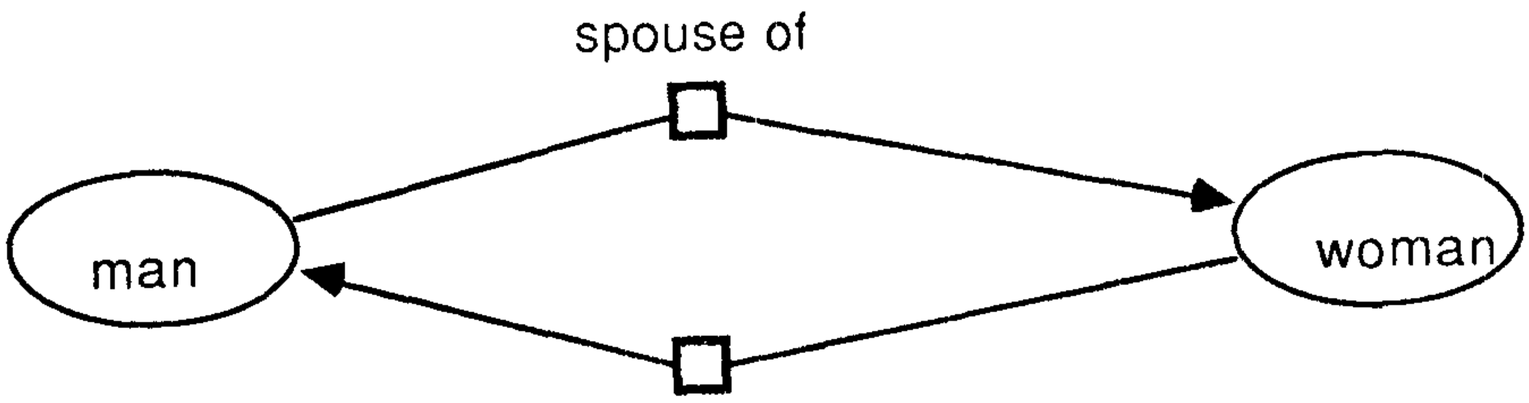
X 좌표를 구하는 과정은 다음과 같다.

```
FOR node IN knowledge base DO layout-X(node);

PROCEDURE layout-X(node);
  BEGIN
    IF node is not layouted yet THEN
       IF node has no child THEN X-pos[node] = 0 ELSE
       BEGIN
       FOR child IN parents(node) DO layout-X(child);
          X-pos[node] = max{X-pos(child)} + offset
       END
  END;
```

Y 좌표를 구하는 과정은 다음과 같다.

```
last_Y = 0;
FOR node IN roots of knowledge base DO layout-Y(node);

PROCEDURE layout-Y(node);
  BEGIN
  IF node has unlayedout children THEN
       BEGIN
       FOR child IN children(node) DO layout-Y(child);
       Y-pos[node] = average-Y(children(node);
       END
    ELSE
    BEGIN
    Y-pos[node] = last_Y + offset;
    last_Y = Y-pod[node]
    END
```

END




<그림 3.21> 순환 구조의 지식망에 대한 예

위의 배치 알고리즘은 지식망 내에 순환 구조(cycle)가 없다는 가정하에서 짜여져 있다. 만약 지식망이 순수하게 개념(concept)들에 의해서만 이루어져 있다면 순환구조의 지식관계가 존재하지 않게된다. 즉, 개념들의 포함 관계에 의한 연결 망, ISA 링크, 만으로 나타 낼수 있다. 그러나 역할(role) 관계도 추가해서 지식을 표현할 때는 <그림 3.21>와 같이 순환 구조가 지식망 내에 존재하게 된다.

Sphinx는 지식 베이스 내에 역할 관계도 표현하므로 순환 구조의 망이 존재하게된다. 그러나 Grapher 알고리즘은 순환 구조의 지식에 대해서는 배치 알고리즘을 수행 시킬 수 없으므로 이를 고려하여 Sphinx에서는 Grapher 알고리즘에 수정을 가하여 사용하였다.

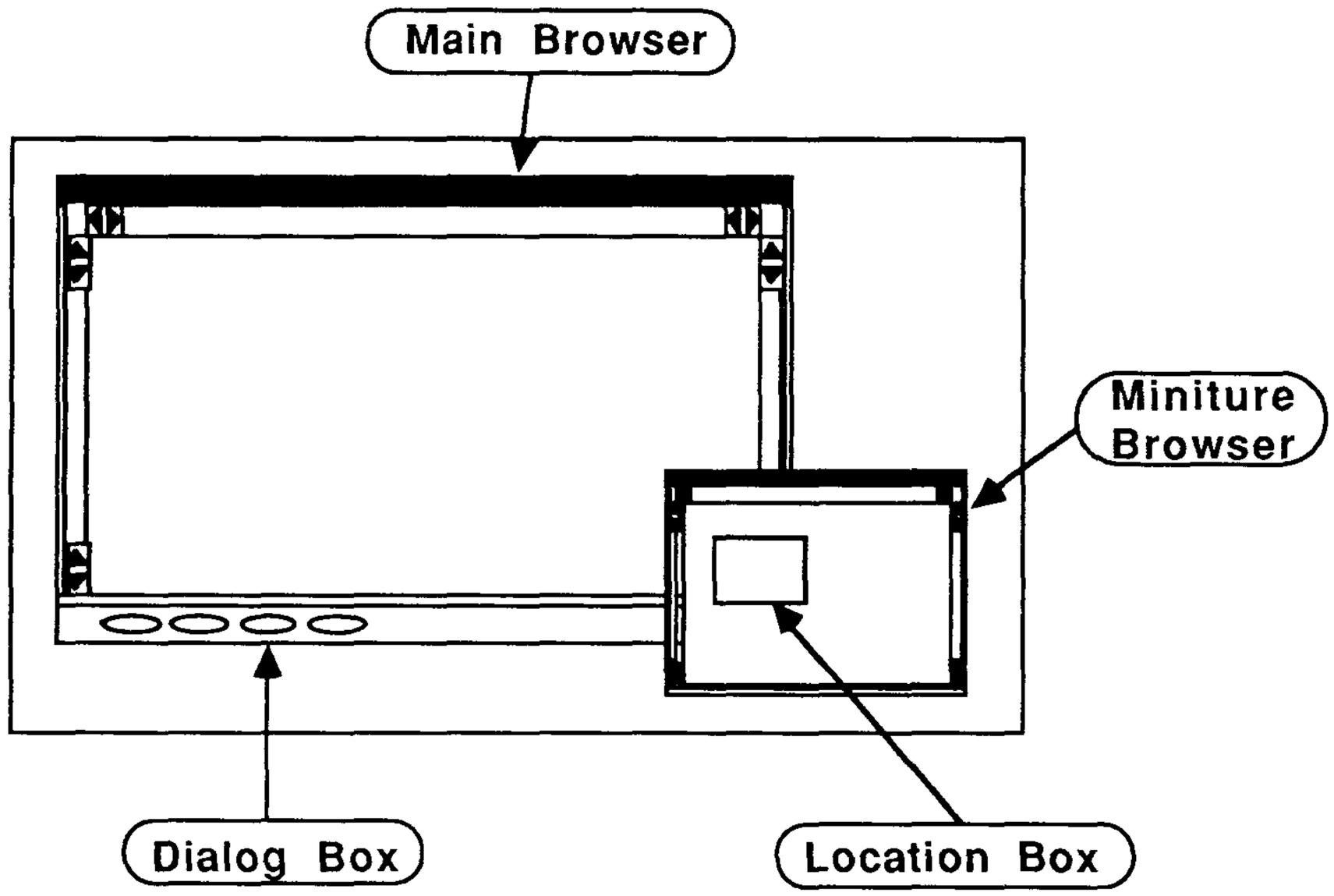
지식망으로부터 개념 지식과 ISA 링크를 추출하며 역할 링크에 대해서는 역할에 대한 지식은 빼고 단지 링크의 존재만을 알리고 순환구조가 발생하는가를 검사해 만약 그러하면 역할 링크의 방향을 바꾸어서 새로운 새로운 지식망을 구성한다. 이는 지식망 내에 순환 구조를 없애고 동시에 역할에 대한 좌표를 연관된 개념 지식 부근에 두고자 하려는 의도에서 취해졌다. 그리고 역할 지식은 디스플레이 과정에서 역할관계가 존재하는 개념 부근에 배치하게된다.

### 3.3.2.2 화면 표시

지식 베이스 내의 각 개념은 윈도우 상의 개념 상자로 표시되며, 개념들 사이의 계층적 구조는 링크로 표시되어 진다. 개념 상자는 상자 내부에 개념 이름이 표시되어 있어 각 개념을 인식할 수 있도록 되어 있다. 링크는 왼쪽에 연결된 개념이 오른쪽에 연결된 개념의 상위 개념임을 의미한다. 역할 관계는 영역 (domain) 개념과 범위 (range) 개념 사이에 정의되므로, 이를 표시하기 위하여 방향성 링크를 사용한다. 즉, 방향 링크의 출발점은 영역 개념을 나타내고 끝점은 범위 개념을 나타낸다. 그리고 역할 이름은 링크

상에 표시되어 진다. 본 브라우저에서는 계층적 구조를 나타내는 링크 ("is" 링크)와 역할 링크를 구별하기 위하여, "is" 링크는 굵은 실선으로, 역할 링크는 가는 실선으로 표시한다.

브라우저에 관한 모든 작업은 마우스를 이용하여 브라우저 윈도우 상에서 행하여지는데, 사용자에게 보다 편의를 제공하기 위해 <그림3.22>과 같은 세개의 부윈도우(subwindow)로 구성된다.




<그림 3.22> Sphinx 브라우저의 화면 구성

a) Main 브라우저

이 브라우저를 통하여 현재 정의된 지식 및 그 관계를 알아볼 수 있다. 화면 상에 나타나는 개념 및 역할은 전체 지식 베이스의 일부분에 불과하므로 지식 베이스의 전체적인 구조를 파악하려면 윈도우 시스템에서 제공하는 scrolling bar를 이용하거나 miniature 브라우저에서 제공하는 scrolling 기능을 이용하여야 한다.

Main 브라우저에서의 모든 작업은 마우스를 이용하게 되는데, 마우스의 커서 위치가 개념을 나타내는 상자 내에 위치하면 이 상자와 이 개념에 상응하는 miniature 브라우저의 상자가 인버스되게 된다. 이 상태에서 마우스의 버튼을 누르면 이 개념에 관한 정의, 실예(instance) 및 역할에 관련된 작업을 수행할 수 있는 Popup menu가 나타난다. 이 popup menu에서 원하는 작업을 선택하면 해당 작업의 결과가 브라우저 상에 나타난다. 그리고

마우스를 원하는 개념에 위치시키고 마우스를 누른 상태로 원하는 위치로 마우스를 이동시키면 선택된 개념 상자 및 이에 상응하는 miniature 브라우저내의 상자가 이동하게 된다.

b) Miniature 브라우저

지식 베이스에 정의된 지식 모두를 한 눈에 볼 수 있는 것이 이상적이겠으나 지식 베이스의 양이 크거나 main 윈도우의 제한된 크기로 인하여 화면 상에 보여줄 수 있는 지식의 양이 제한을 받게 된다. 이러한 문제점을 극복하고 지식 베이스의 조감도 (bird's eye view)를 제공하기 위해서 지식 베이스의 miniature를 제공한다.

miniature는 지식 베이스의 보다 많은 관계를 나타냄으로써 사용자에게 지식 베이스의 내용을 보다 포괄적으로 제공하게 되나 대신에 자세한 내용은 파악하기 힘들다. 본 브라우저에서는 miniature 브라우저내의 location 상자를 이용하여 이러한 문제점을 해결하였다. 즉, location 상자를 마우스로 선택하여 miniature 브라우저에서 이동하게 되면 이 location 상자 내부에 포함되는 지식 베이스의 구조적 관계가 main 브라우저에 나타나게 된다.

miniature 브라우저에서도 main 브라우저에서와 유사한 방법으로 개념을 선택하고 이를 원하는 위치로 이동시킬 수 있다. 이외에, 개념 상자에서 마우스의 오른쪽 버튼을 누르면 해당 개념의 이름이 표시되며, 오른쪽 버튼을 누르면 이 개념에 상응하는 main 브라우저의 상자가 main 브라우저의 중심에 위치하게 된다.

c) Dialog 상자

Sphinx 시스템은 개념들 간의 계층적 관계 및 이들 개념 사이의 역할 관계를 모두 표현할 수 있다. 브라우저에서는 보다 좋은 인터페이스를 제공하기 위하여 두 종류의 브라우저, 계층적 관계를 나타내는 브라우저와 역할 관계를 포함하는 브라우저,를 제공하고 있다. dialog 상자에는 이 두 브라우저를 선택하는 기능 및 브라우저에서 빠져나와 Sphinx 시스템으로 되돌아 가는 기능을 제공한다.

Sphinx 시스템에서 브라우저를 호출하면 기본적으로 현 지식 베이스에 대한 계층적 구조를 보여주게 되는데, 이 상태에서 dialog 상자 내에 있는 역할 표시 버튼을 누르면 역할 관계 구조가 표시되게 된다. 그리고 역할 제어 버튼을 누르면 역할 관계 구조가 사라지고 원래의 계층적 구조만 main 브라우저에 남게된다.

### 3.3.3 브라우저의 사용 예

브라우저를 사용하려면 먼저 지식 편집기나 일반 문서 편집기를 사용하여 지식 베이스를 구축하여야 한다. 예를들어 일반 문서 편집기를 사용하여 아래와 같이 가족 관계를 지식 베이스로 구축하였다고 하자.

```
defConcept(person).
defConcept(male, person).
defConcept(female, person).
defRole(child, person, person).
defRole(spouse, person, person).
defConcept(married, person).
defConcept(parent, person).
defConcept(father, [male, parent]).
defRole(son, child, person, male).
defRole(daughter, child, person, female).
defConcept(mother, [female, parent]).
defConcept(married_female, [married, female]).
defConcept(successful_father, father, daughter, married_female).
```

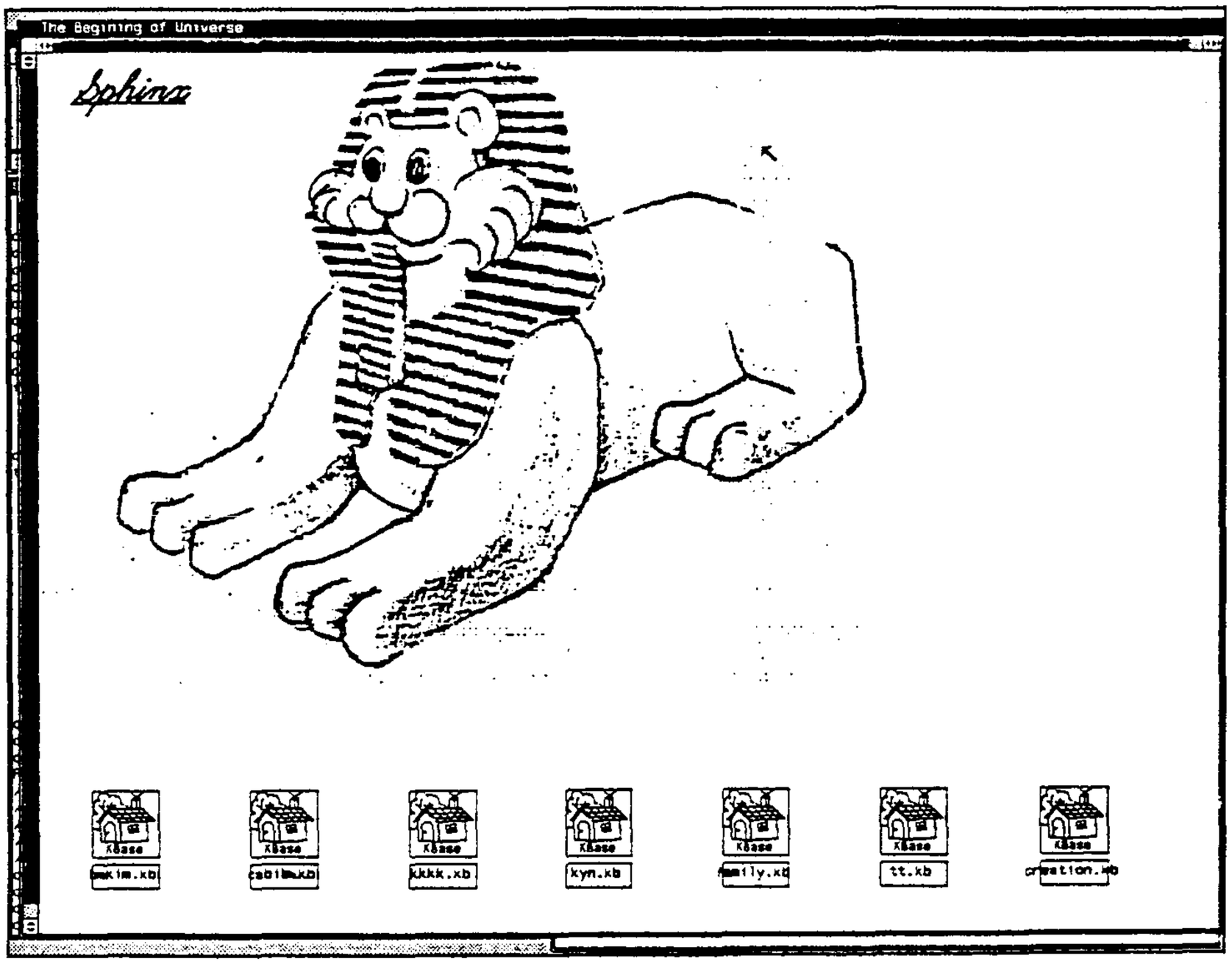
이 상태에서 브라우저를 호출하면, 브라우저는 각 개념의 위치를 계산하여 <그림 3.23>와 같이 화면에 표시하게 된다.

Main 브라우저에서 마우스를 개념 상자위에 위치시키면 <그림 3.24>과 같이 해당 개념 상자와 이에 상하는 miniature 브라우저내의 개념 상자가 검게 변하게 된다. "male"이라는 개념의 실예, 즉 현 지식 베이스에 남자로 기록되어 있는 사람의 명단을 보고 싶으면 마우스를 이용하여 커서를 "male"이라는 개념 상자에 위치시키고 마우스 버튼을 누르면 <그림 3.25>과 같이 popup 메뉴가 표시된다. 이때, "INSTANCE"라는 항목을 선택하면 남자의 명단이 <그림 3.26>과 같이 popup 윈도우에 표시된다. 이 popup 윈도우는 "Quit" 버튼을 선택하면 화면 상에서 사라지게 된다. 그리고 "male"의 정의를 보고 싶으면 <그림 3.27>와 같이 popup 메뉴에서 "DEFINITION" 항목을 선택하면 그에 대한 정의가 <그림 3.28>에서 처럼 부윈도우 상에 표시되게 된다.

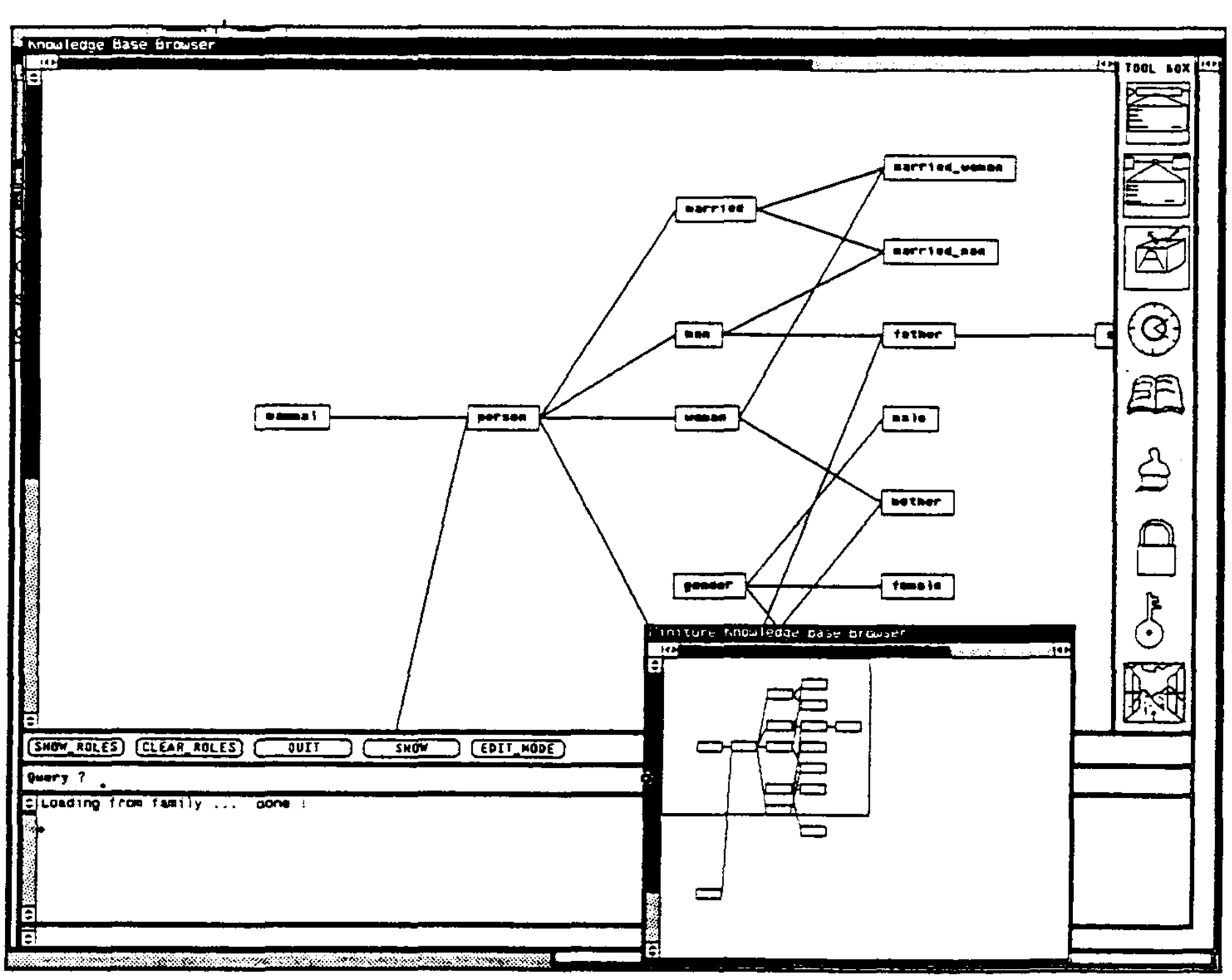
현재 main 윈도우에 표시된 지식 구조는 전체 구조의 일부분에 지나지 않는다. 따라서, main 브라우저 상에 보이지 않는 지식 구조를 보려면 윈도우에 부착된 scrolling

bar나 miniature 브라우저에 있는 location 상자를 이용하여야 한다. <그림 3.23>의 상태에서 location 상자를 <그림 3.29>처럼 이동시키면 location 상자 내에 있는 지식 구조가 main 브라우저에 표시되게 된다. 그리고 <그림 3.30>처럼 miniature 브라우저에서 location 상자 밖에 있는 개념 상자를 마우스의 오른쪽 버튼을 이용하여 선택하면 해당 개념 이름이 표시된다. 따라서 main 브라우저를 변경하지 않고도 그 내부에 표시되지 않는 개념들을 볼 수 있다.

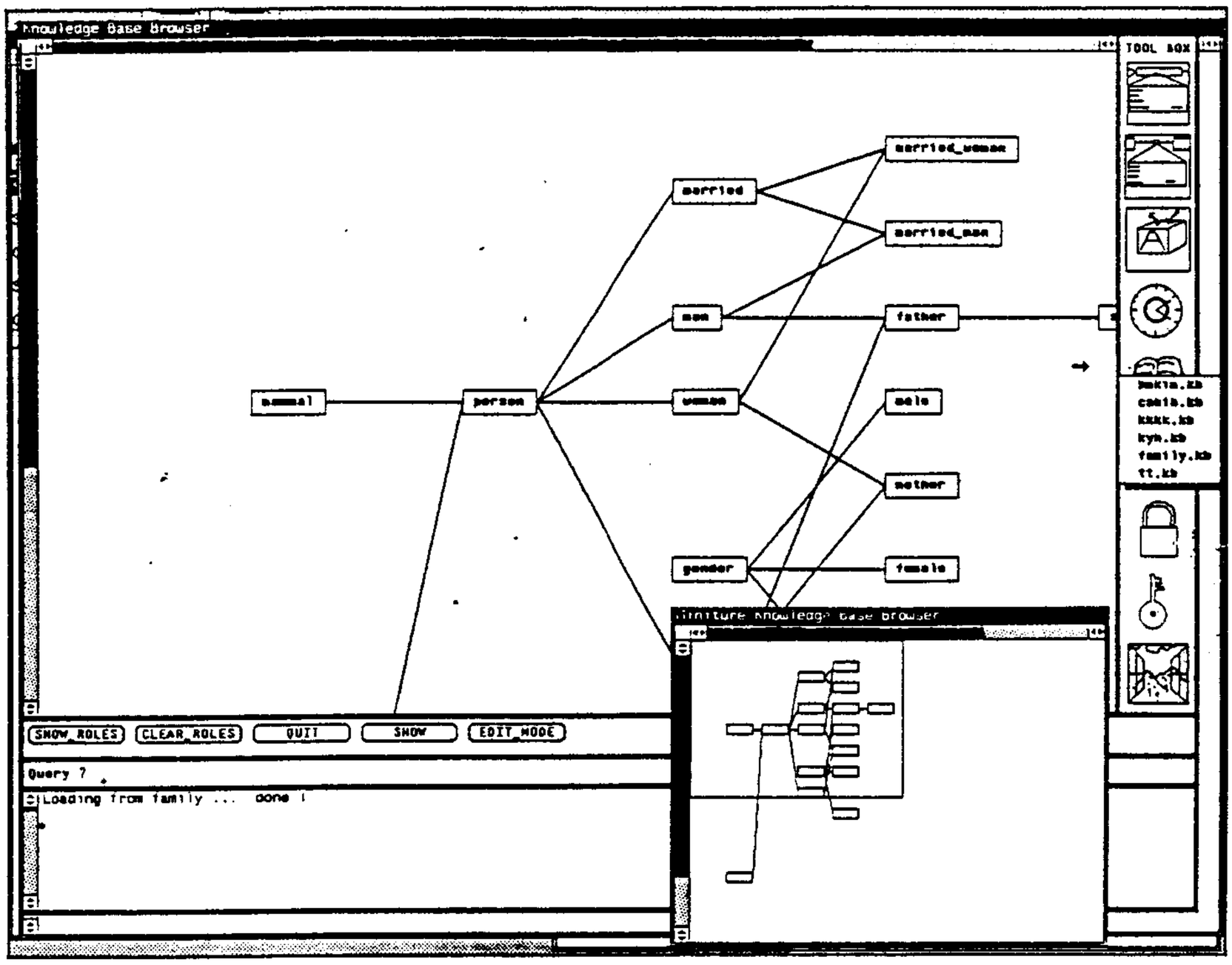
<그림 3.23>의 상태에서 dialog box내의 "SHOW_ROLES"라는 버튼을 선택하면 개념 사이에 정의된 역할 관계가 <그림 3.31>처럼 표시되어 진다. 이 상태에서도 전과 같은 작업을 그대로 수행할 수 있다. 예를들어, <그림 3.32>처럼 location 상자를 이동시키면 그 내부에 있는 지식 구조가 main 브라우저에 표시된다든지, 또는, locatin 상자 밖에 있는 개념 상자를 마우스의 왼쪽 버튼을 사용하여 선택하면, <그림 3.33>처럼 해당 개념이 main 브라우저의 중앙에 위치하게 된다.



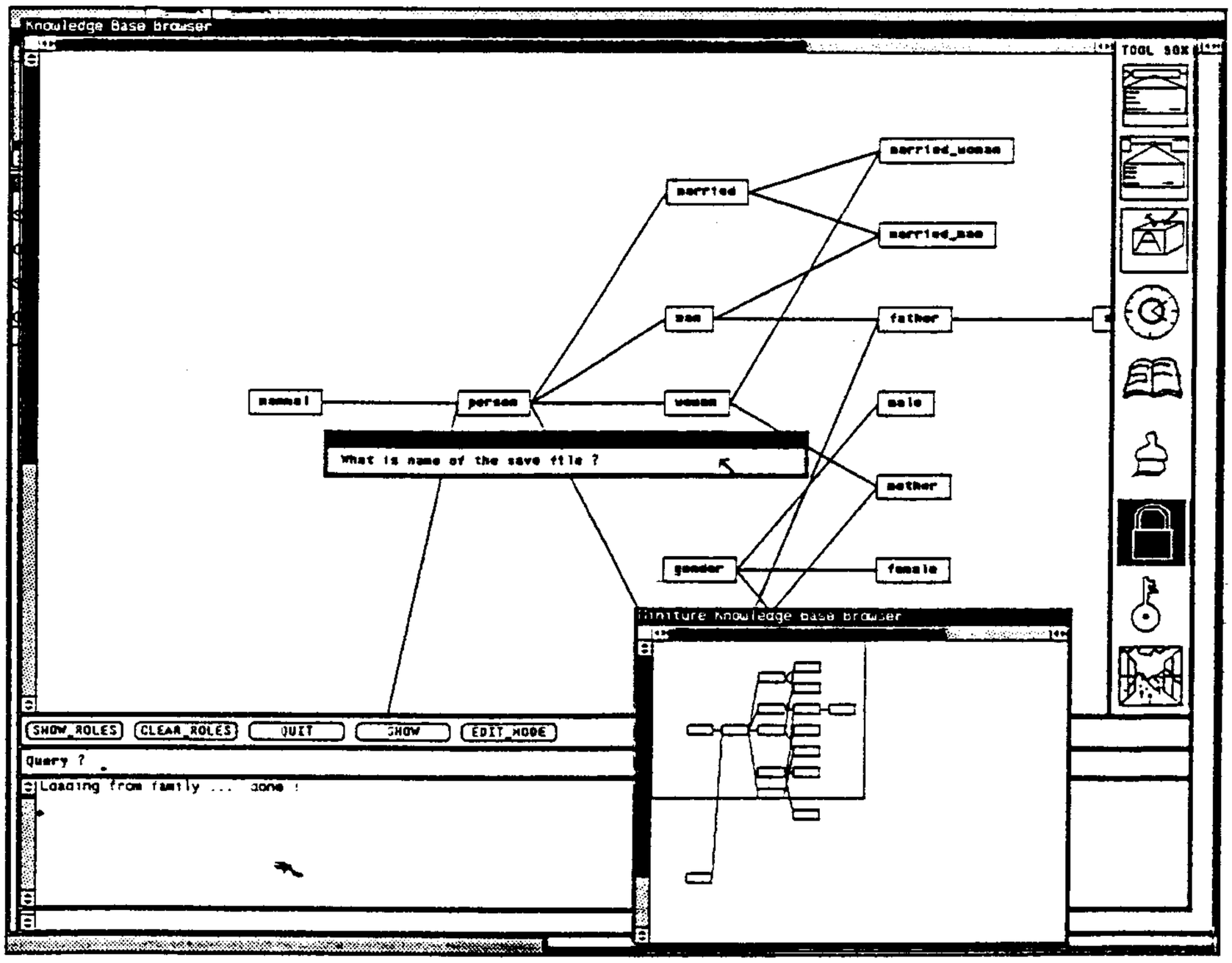

<그림 3.2> Sphinx의 초기 화면



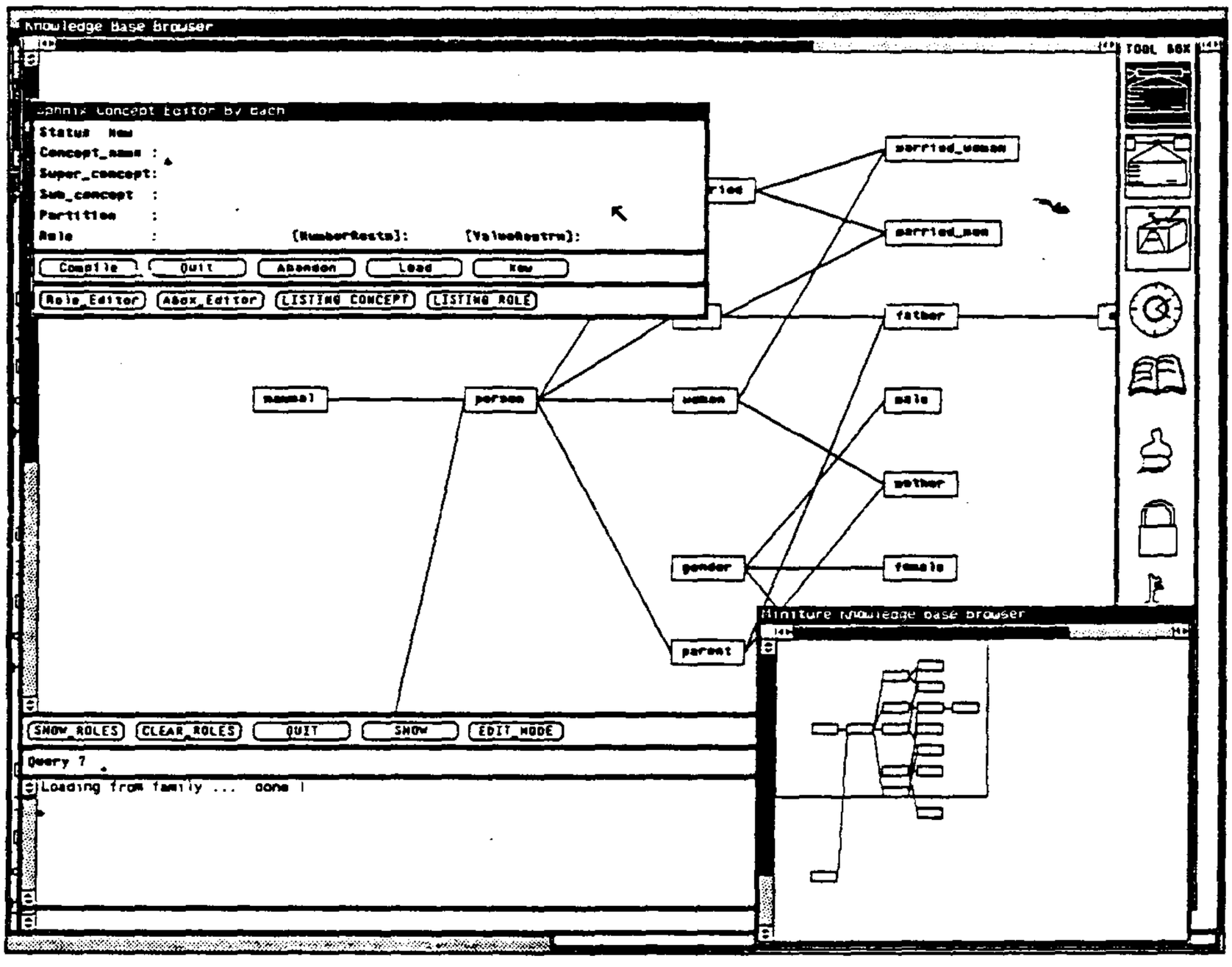

<그림 3.3> 지식베이스가 적재된 후의 Sphinx 화면



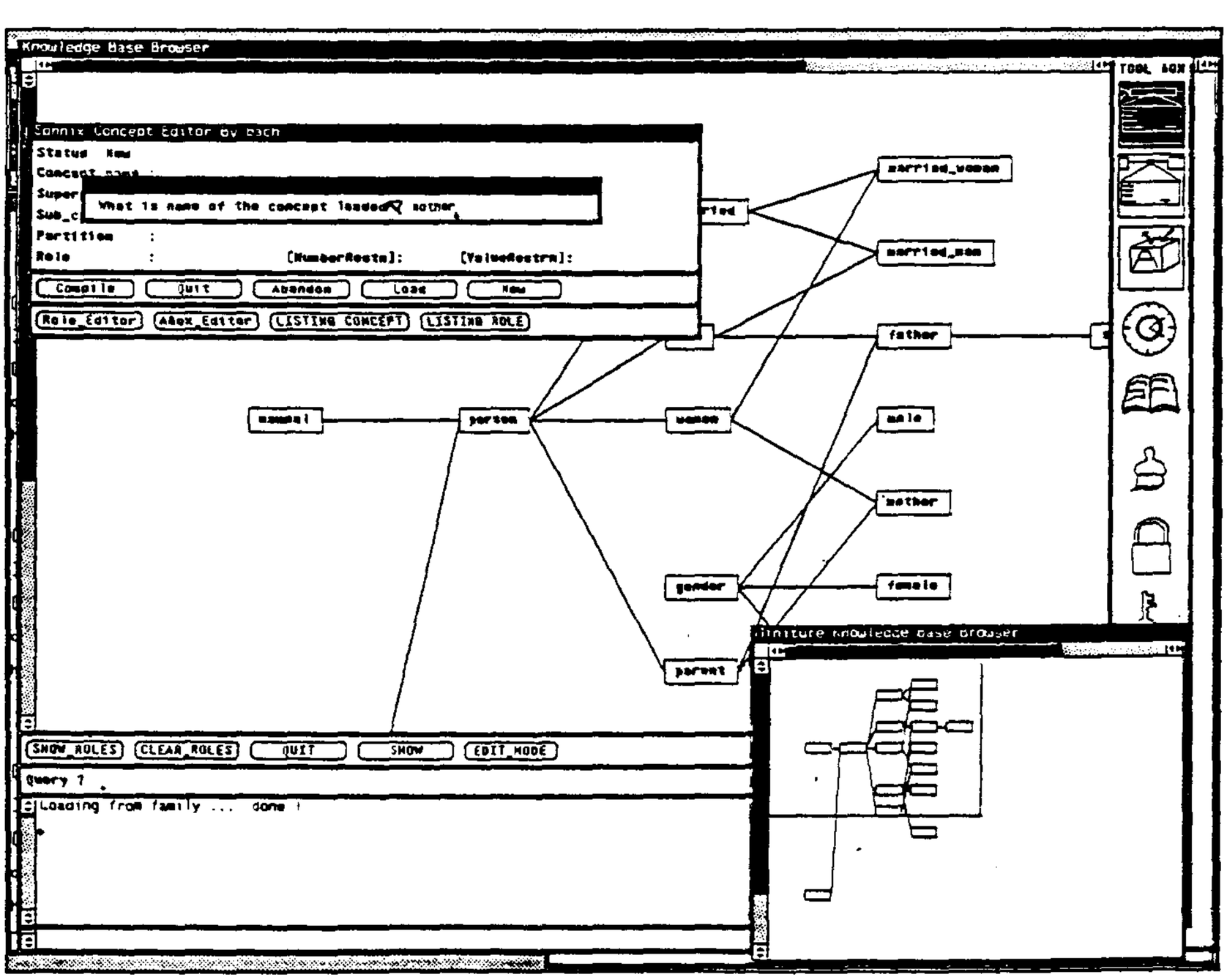

<그림 3.4> 지식 디렉토리 아이콘의 Pop-Up 메뉴



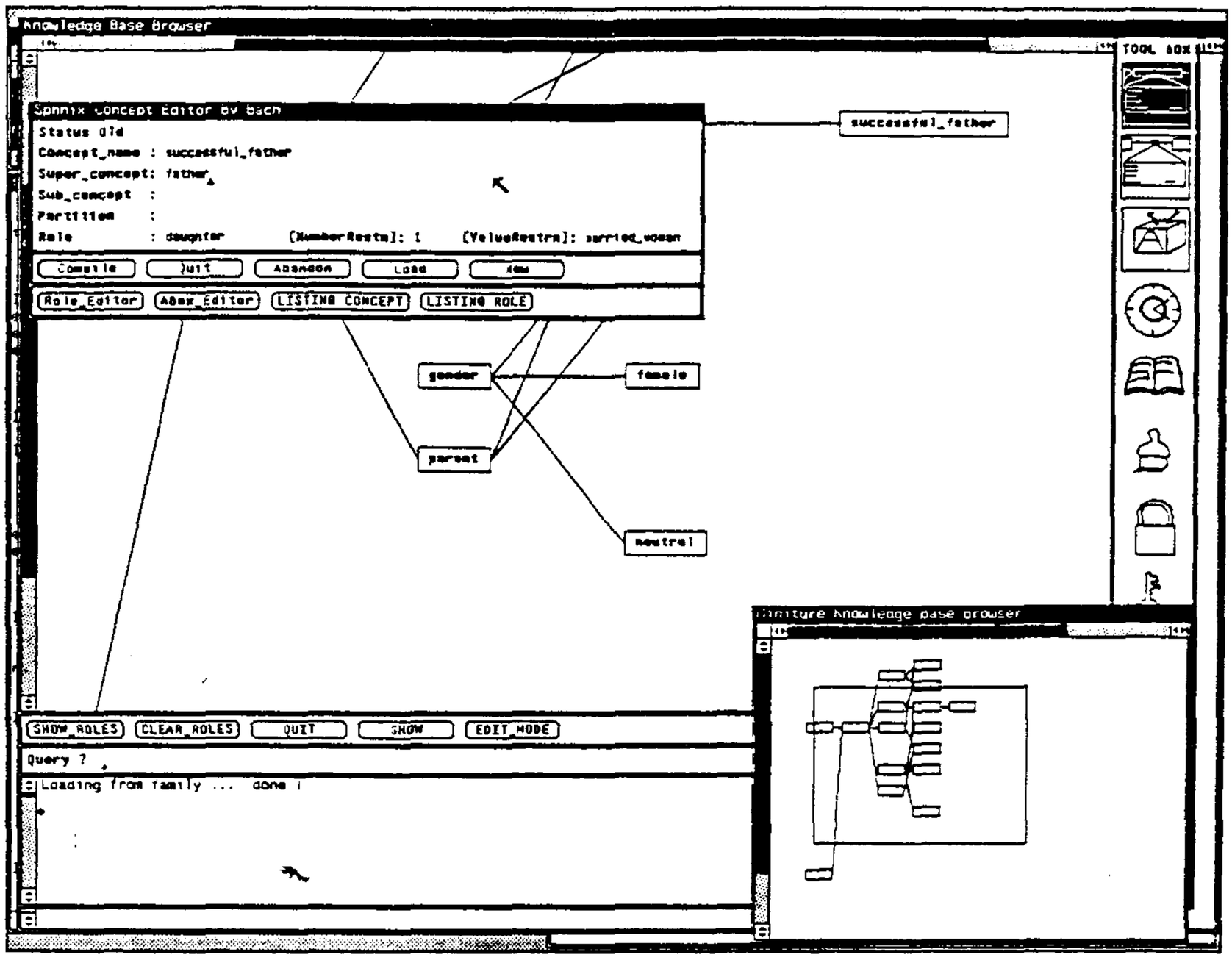

<그림 3.5> 지식베이스 저장 대화 상자



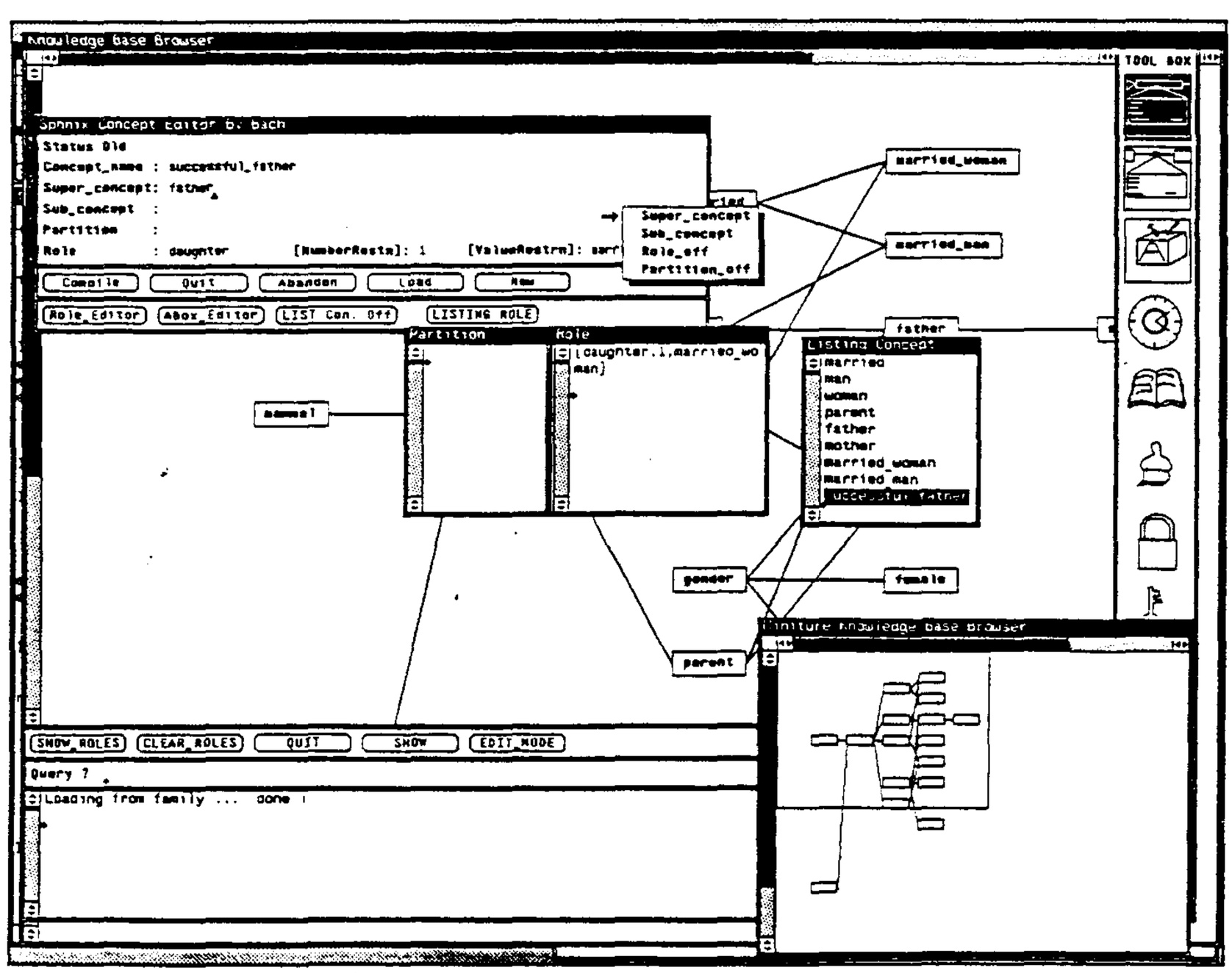

<그림 3.7> 개념 편집기 초기 화면



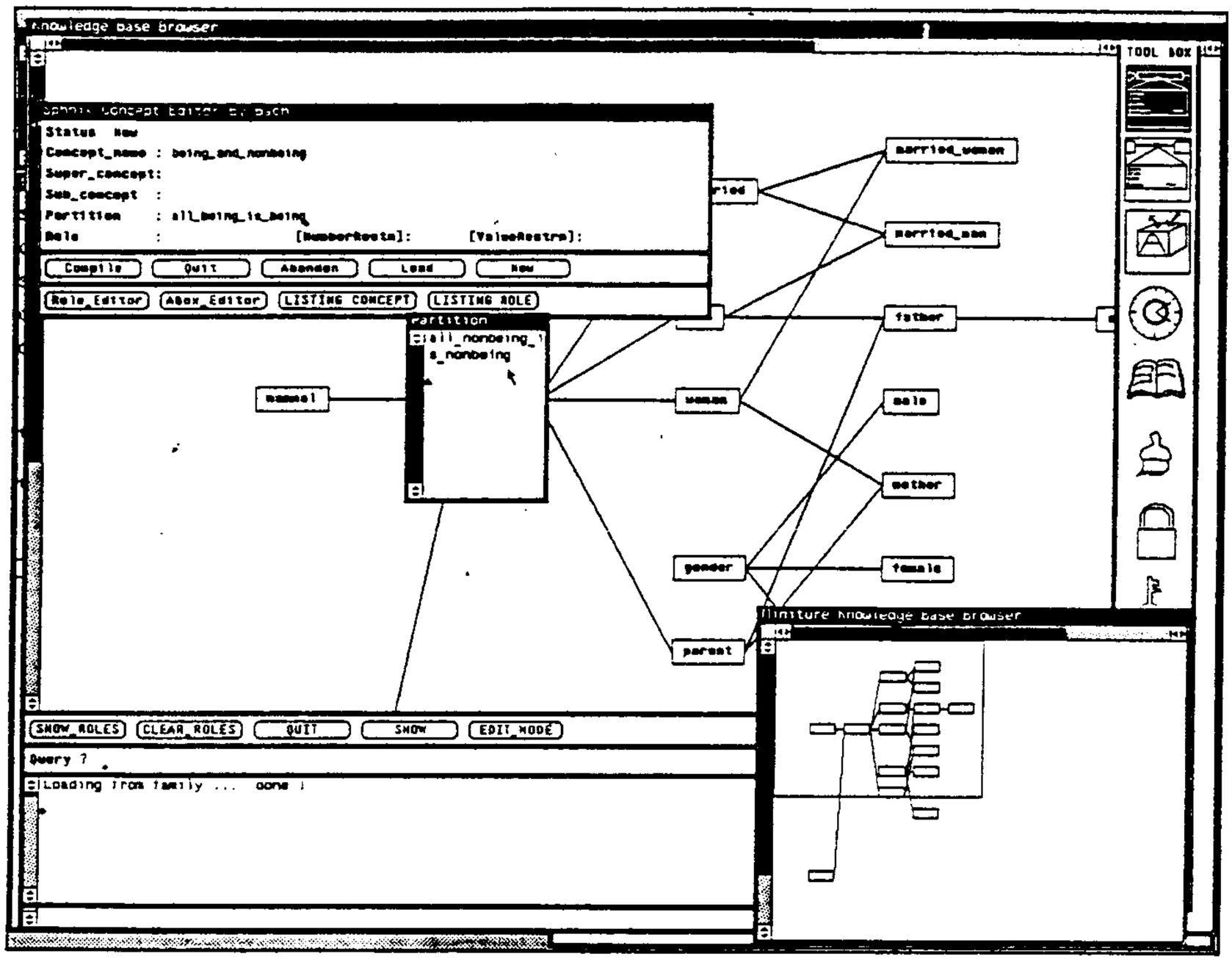

<그림 3.8> 정의된 개념을 보여 주기 위한 대화 상자



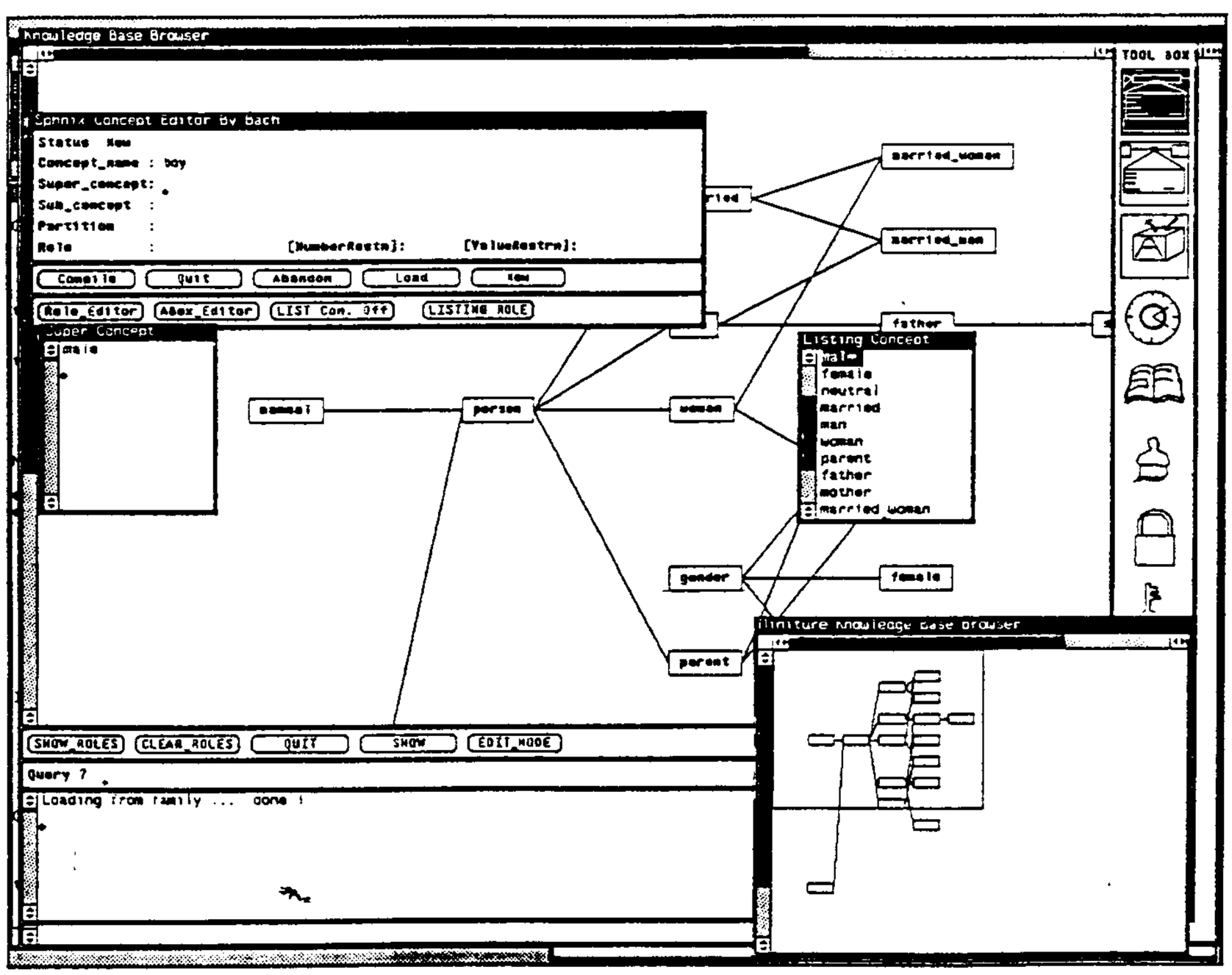

<그림 3.9> 호출된 개념 정의예



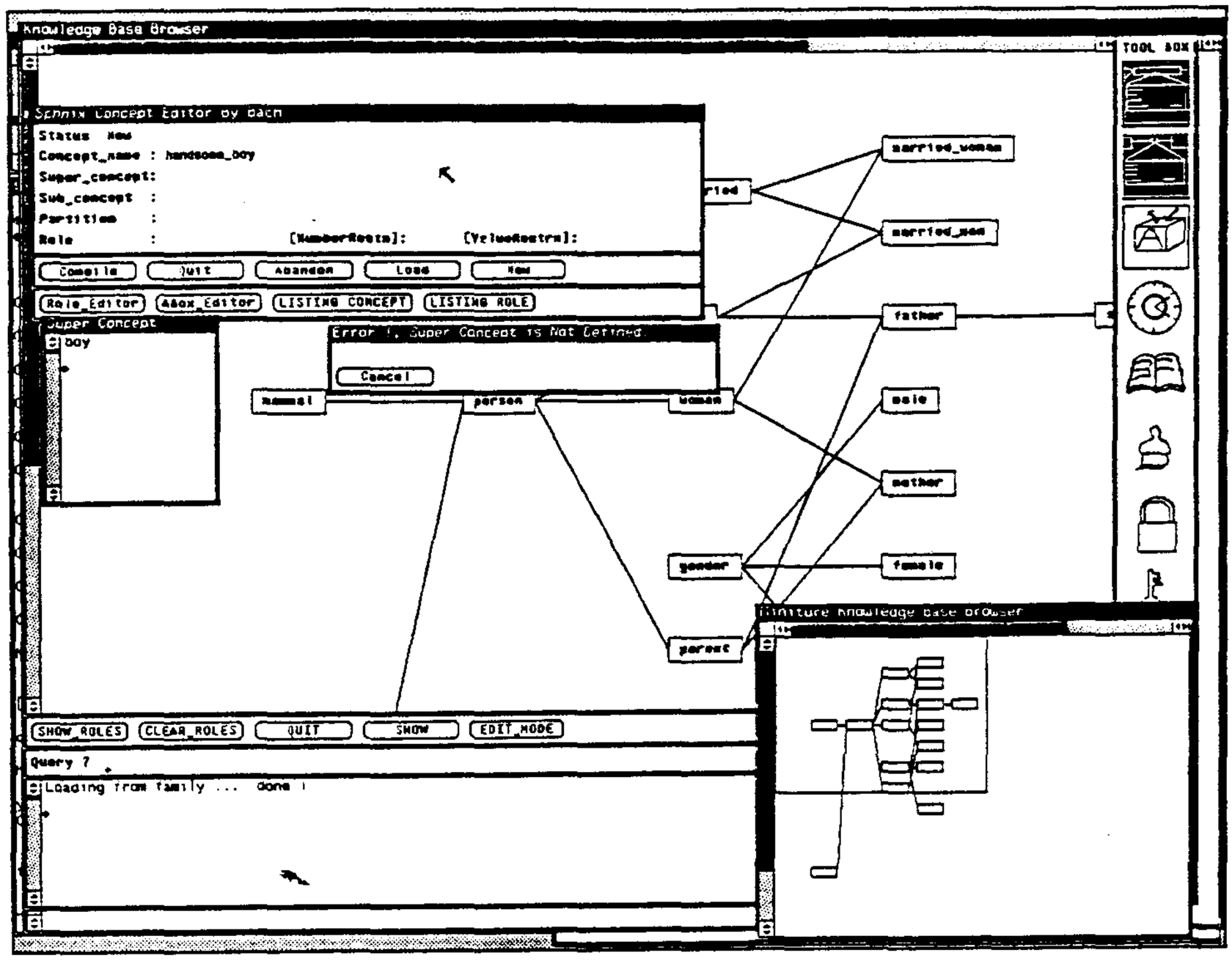

<그림 3.10> 개념 편집기에서의 PopUp 메뉴



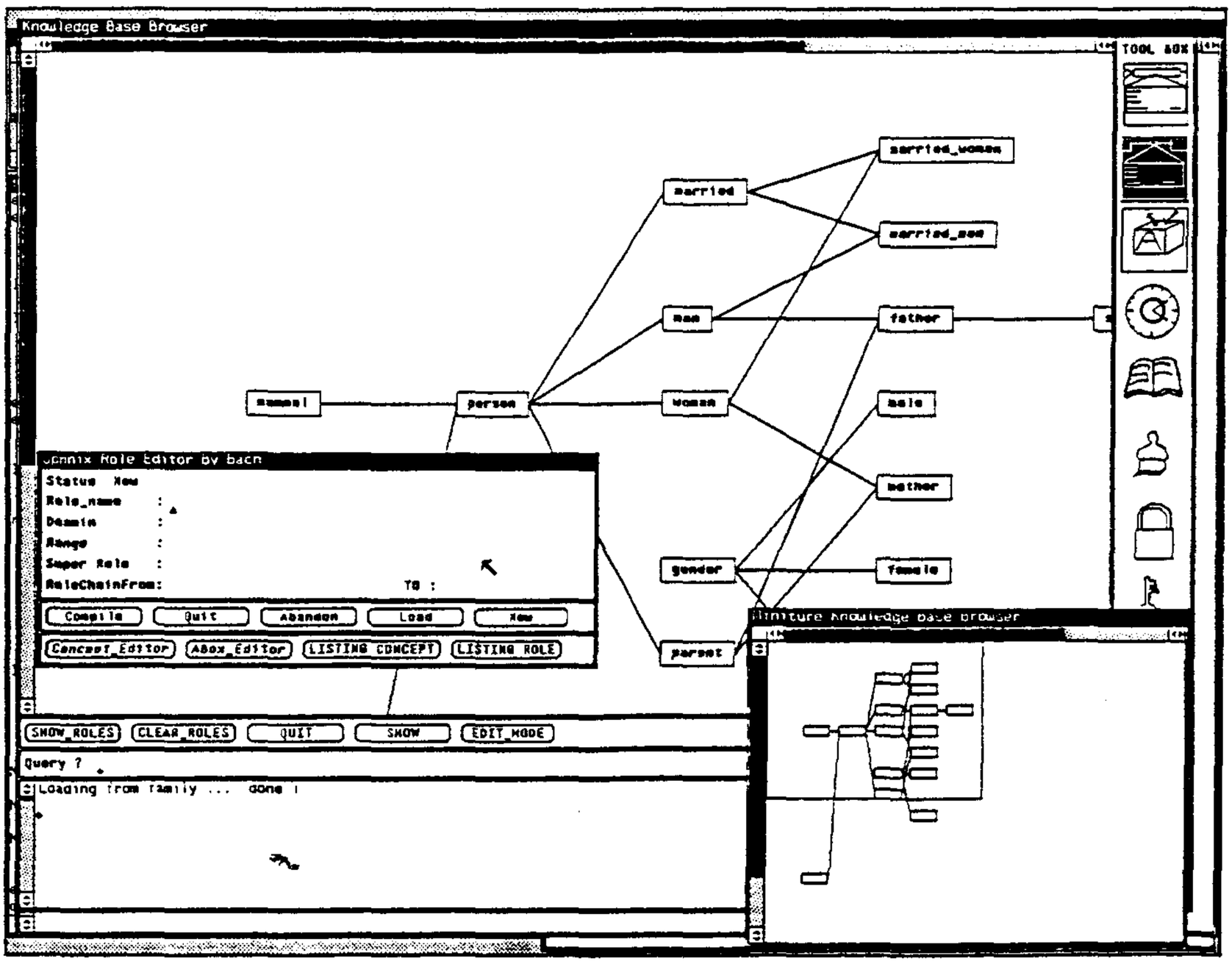

<그림 3.11> 개념 편집기에서 Partition의 한 항목의 수정



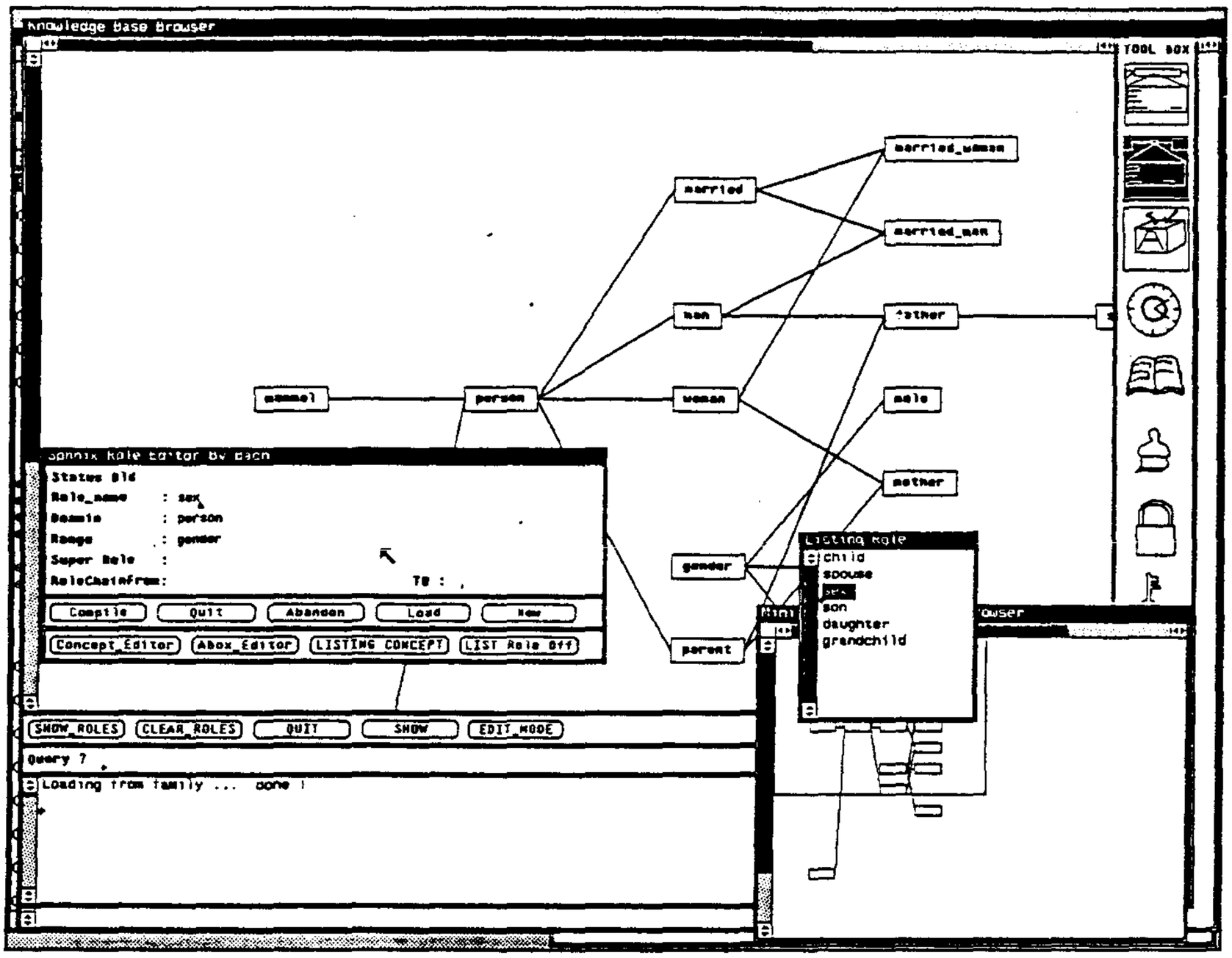

<그림 3.12> 리스팅 기능을 사용한 입력



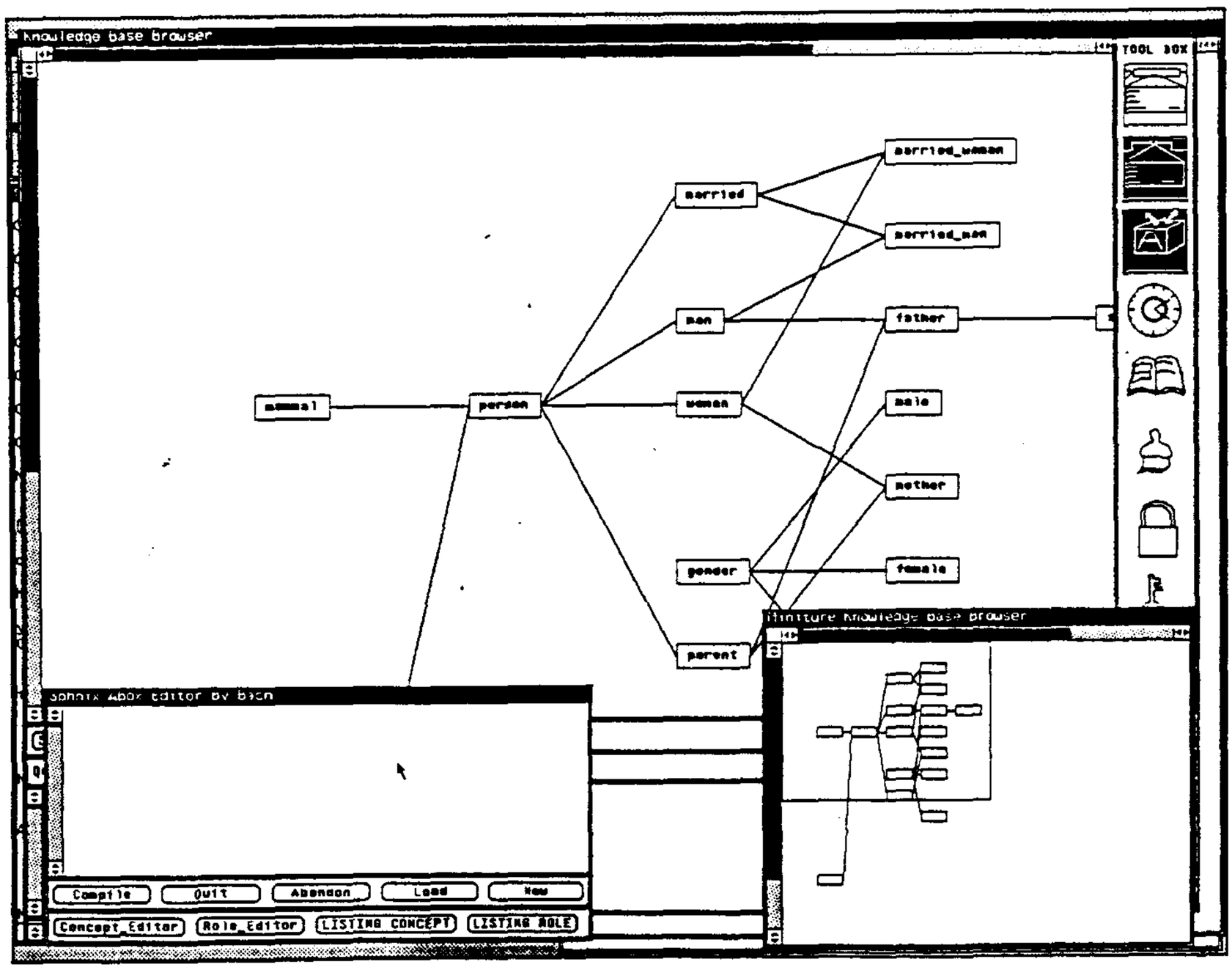

<그림 3.13> 에러가 발생한 경우의 개념 편집기



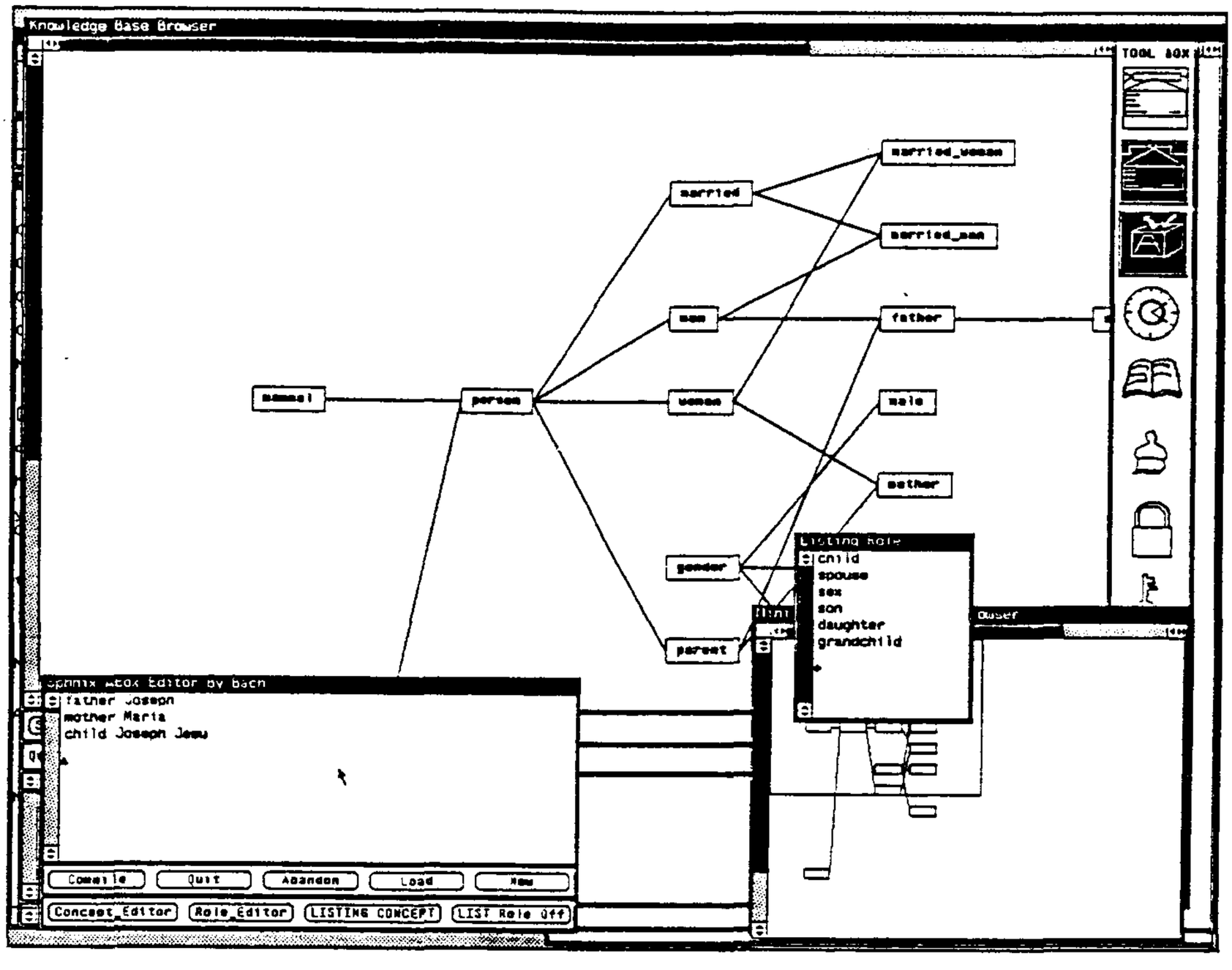

<그림 3.14> 역할 편집기의 초기 화면



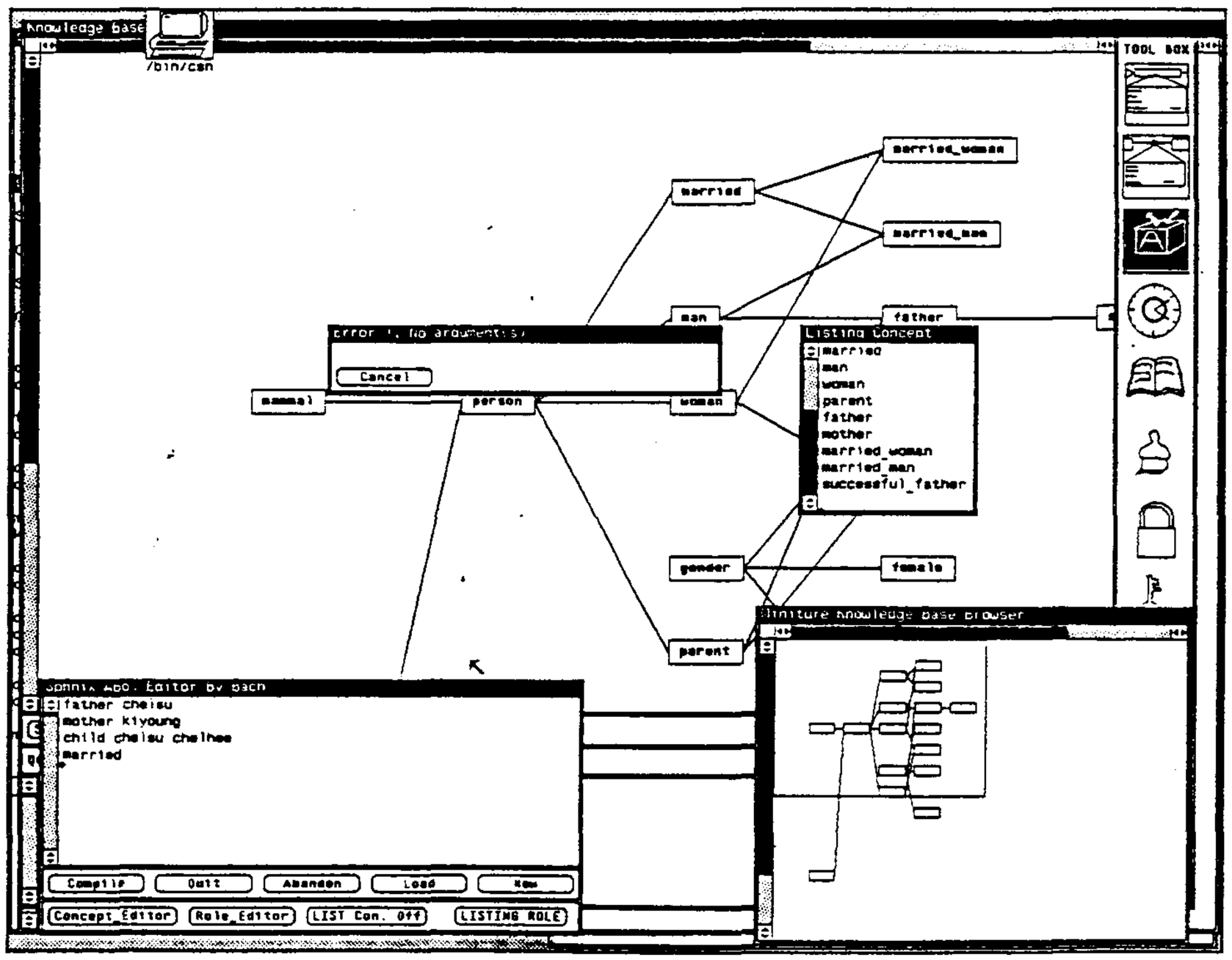

<그림 3.15> 역할 편집기에 보여진 역할의 예



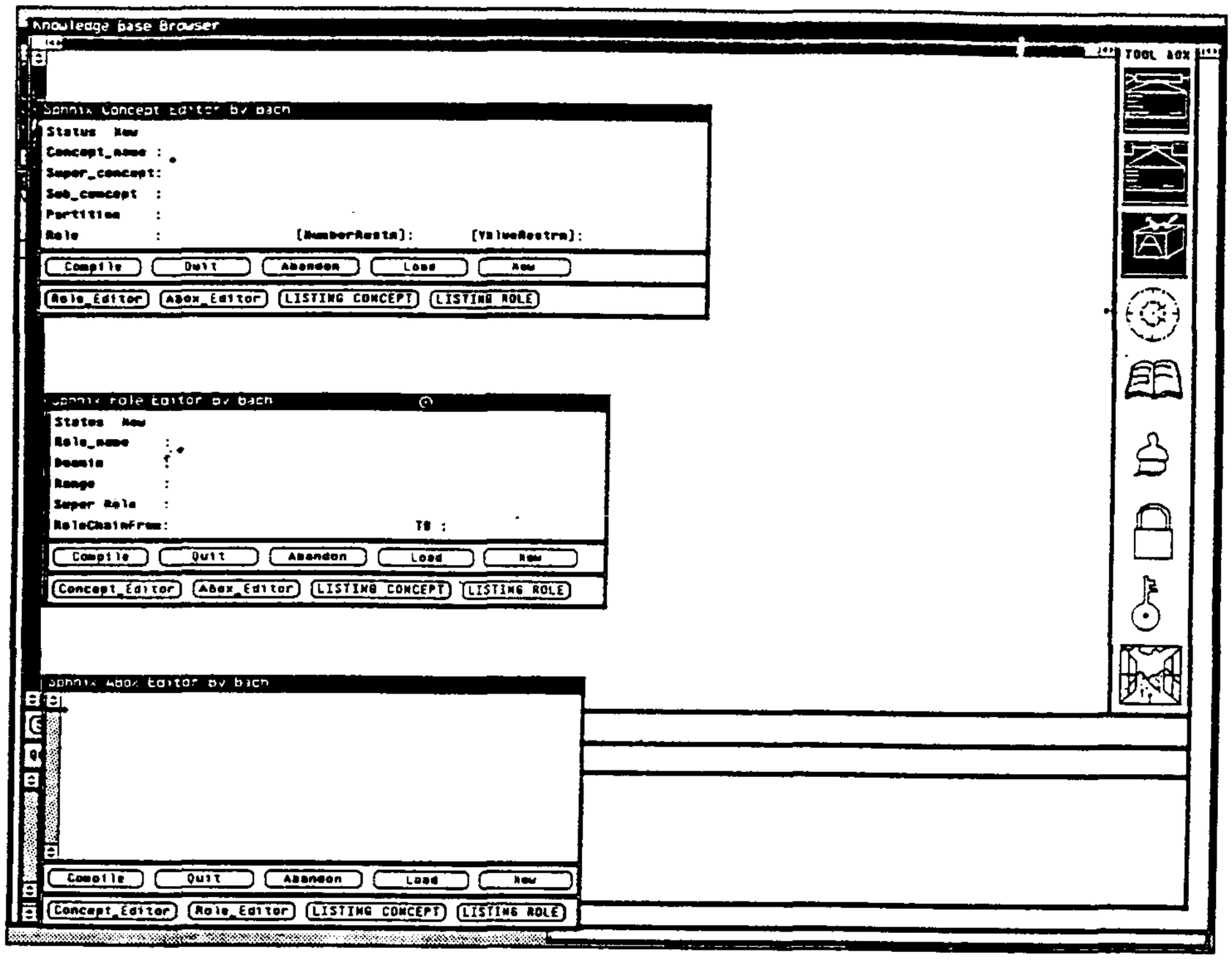

<그림 3.16> ABox 편집기의 초기 화면



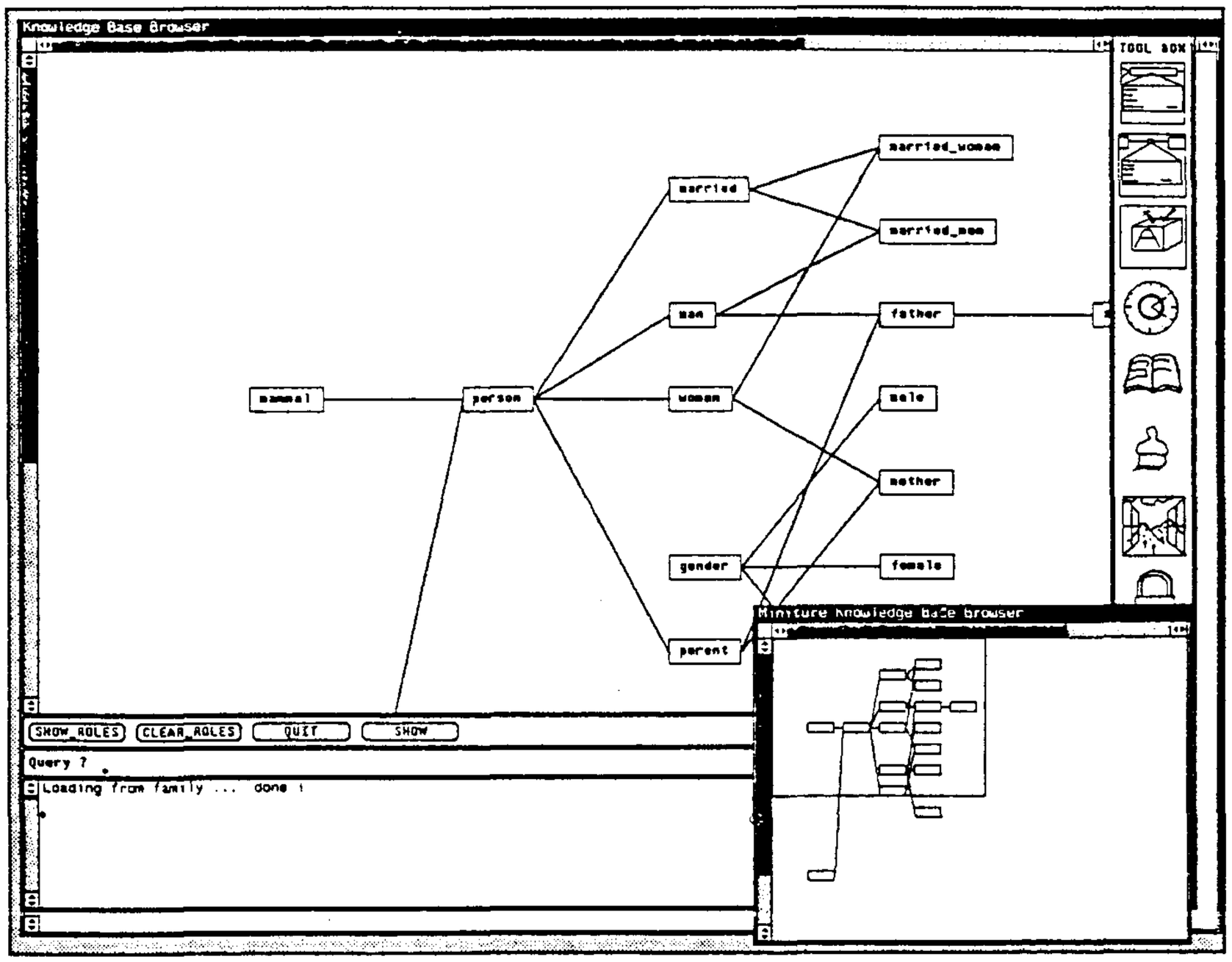

<그림 3.17> ABox 편집기에 보여진 단정적 지식



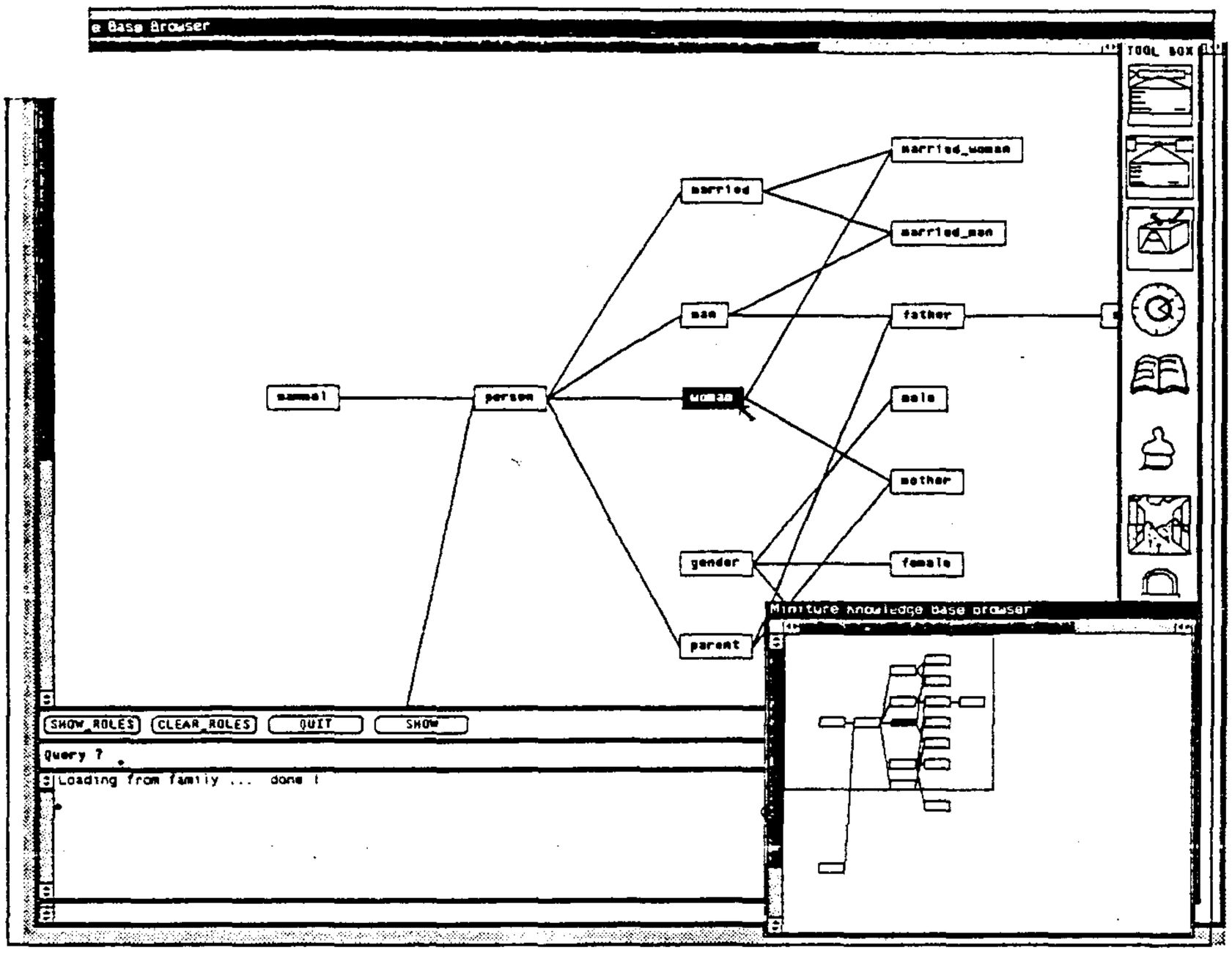

<그림 3.18> ABox 편집기에서 잘못 입력한 경우의 경고 화면



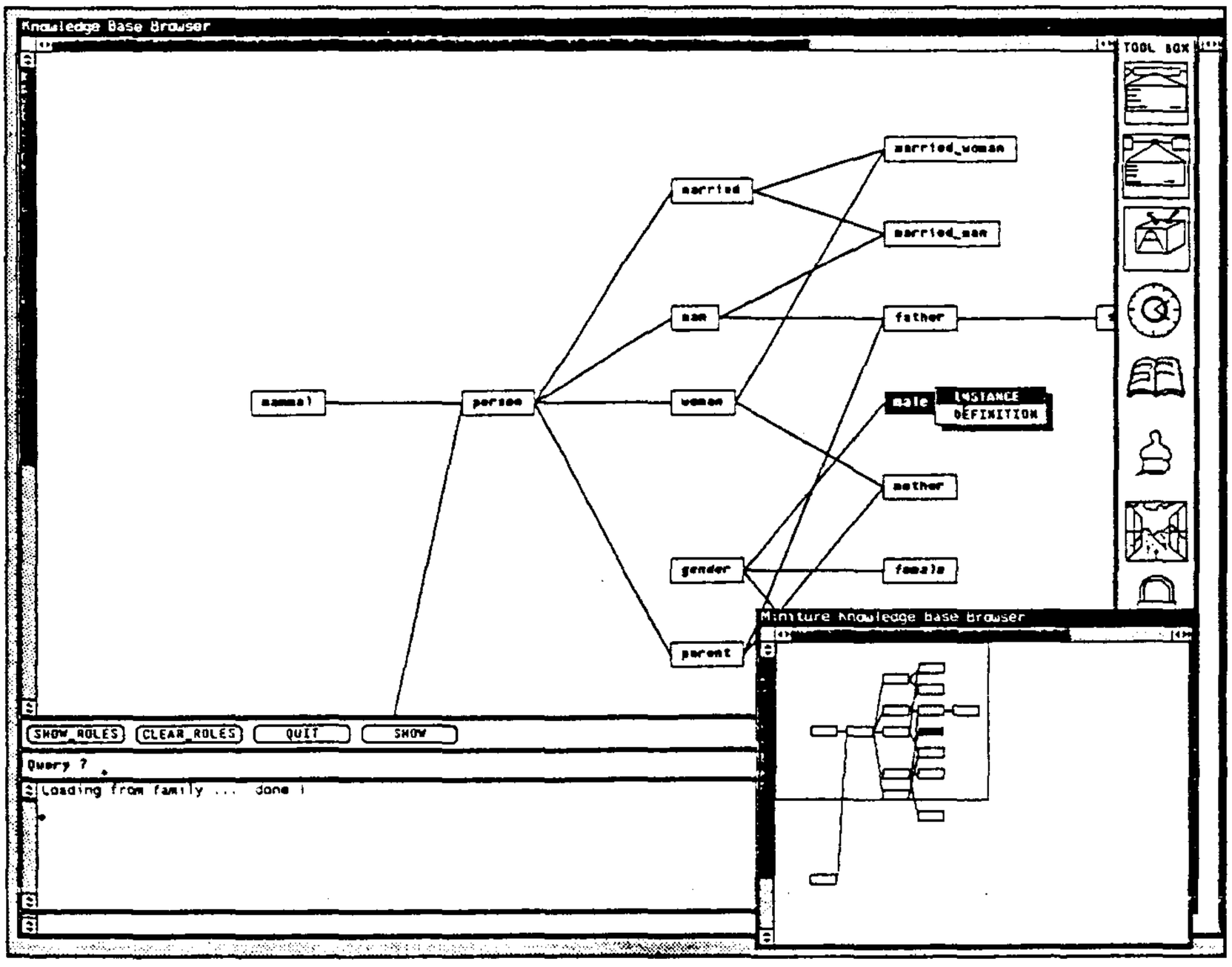

<그림 3.19> 세개의 편집기가 모두 사용된 화면



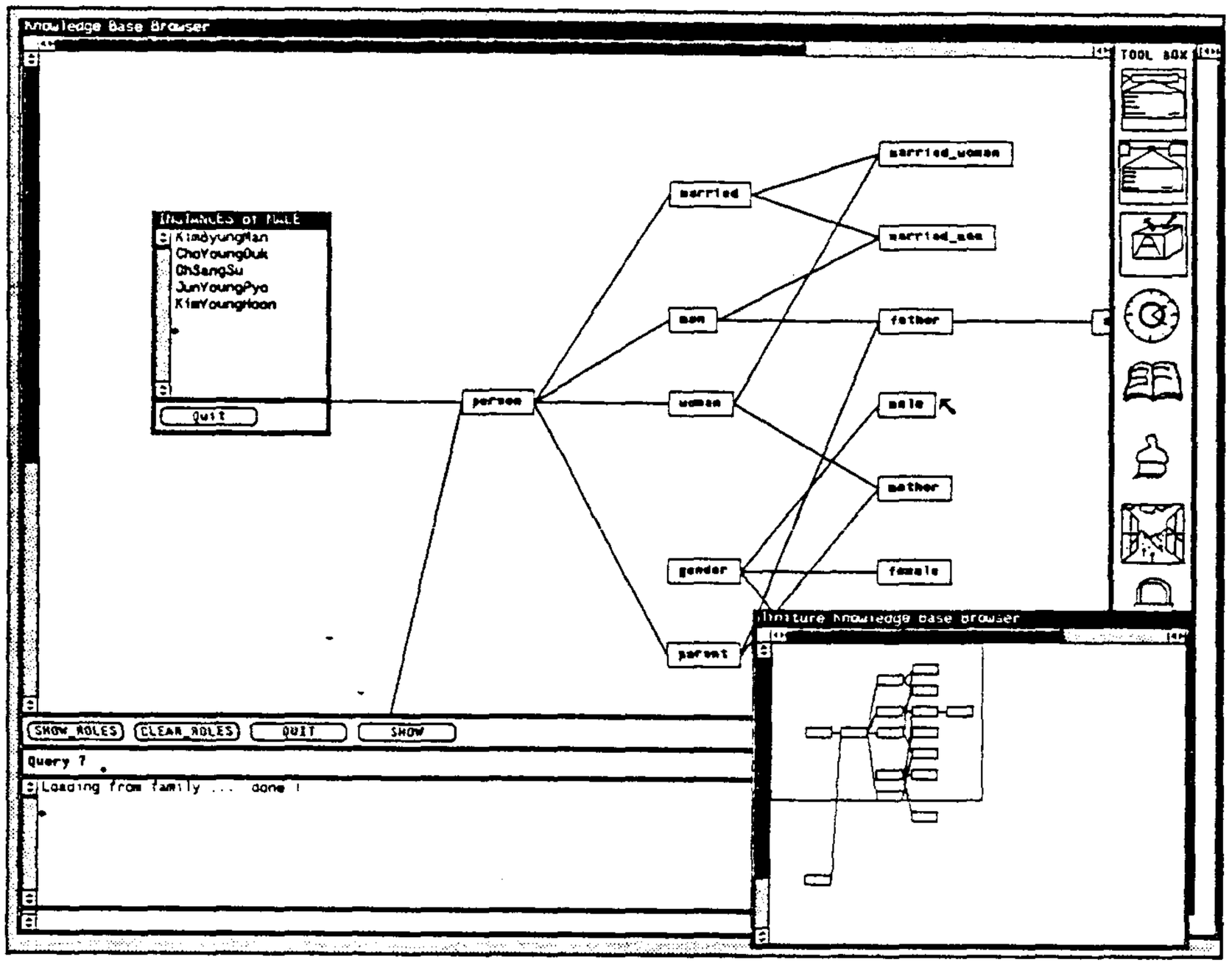

<그림 3.23>


e Base Browser
TOOL BOX
married_woman
married
married_man
man
father
animal
person
woman
male
mother
gender
female
parent
Miniture knowledge base browser
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

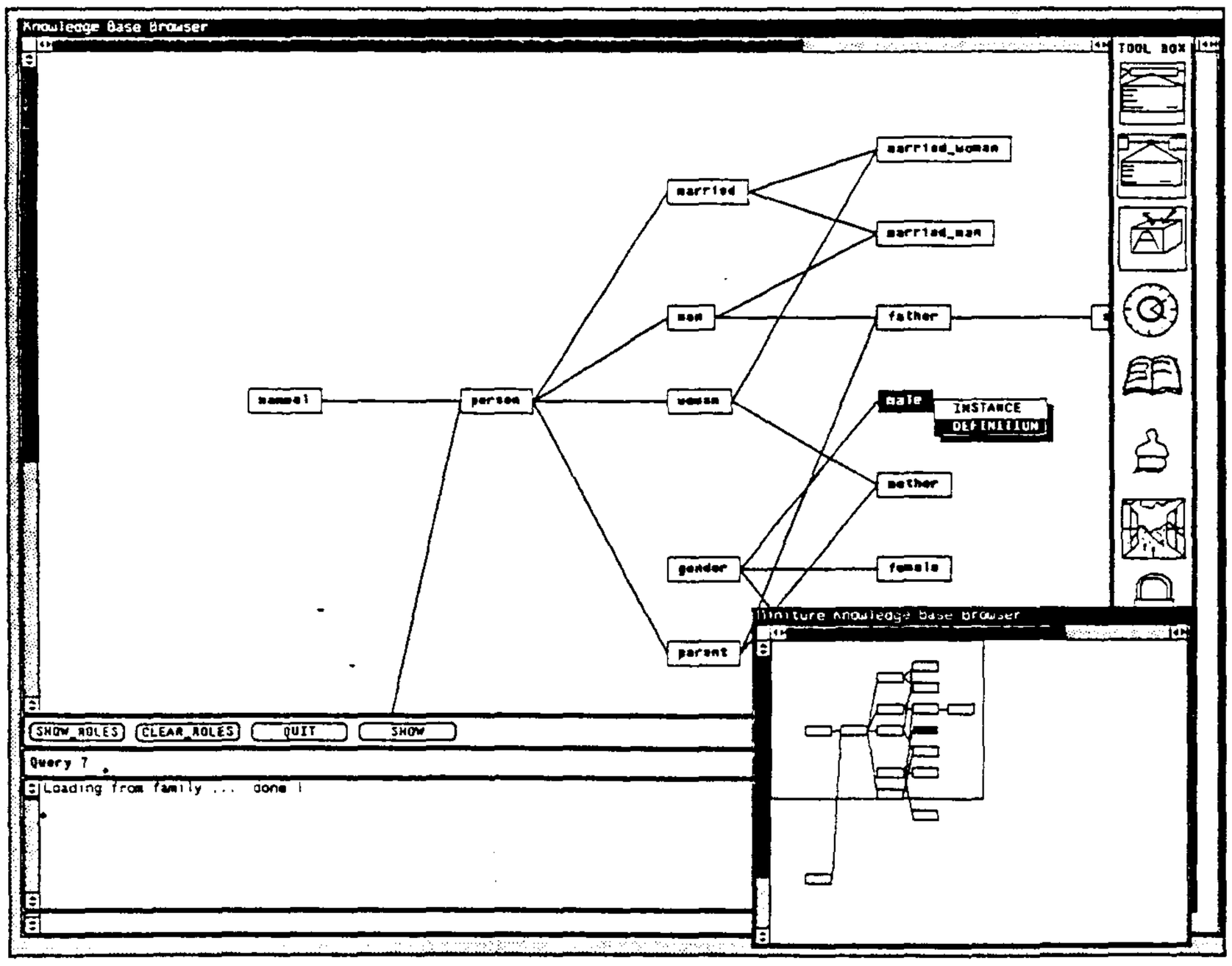
<그림 3.24>


Knowledge Base Browser
TOOL BOX
married_woman
married
married_man
man
father
animal
person
woman
male
INSTANCE
DEFINITION
mother
gender
female
parent
Miniture knowledge base browser
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

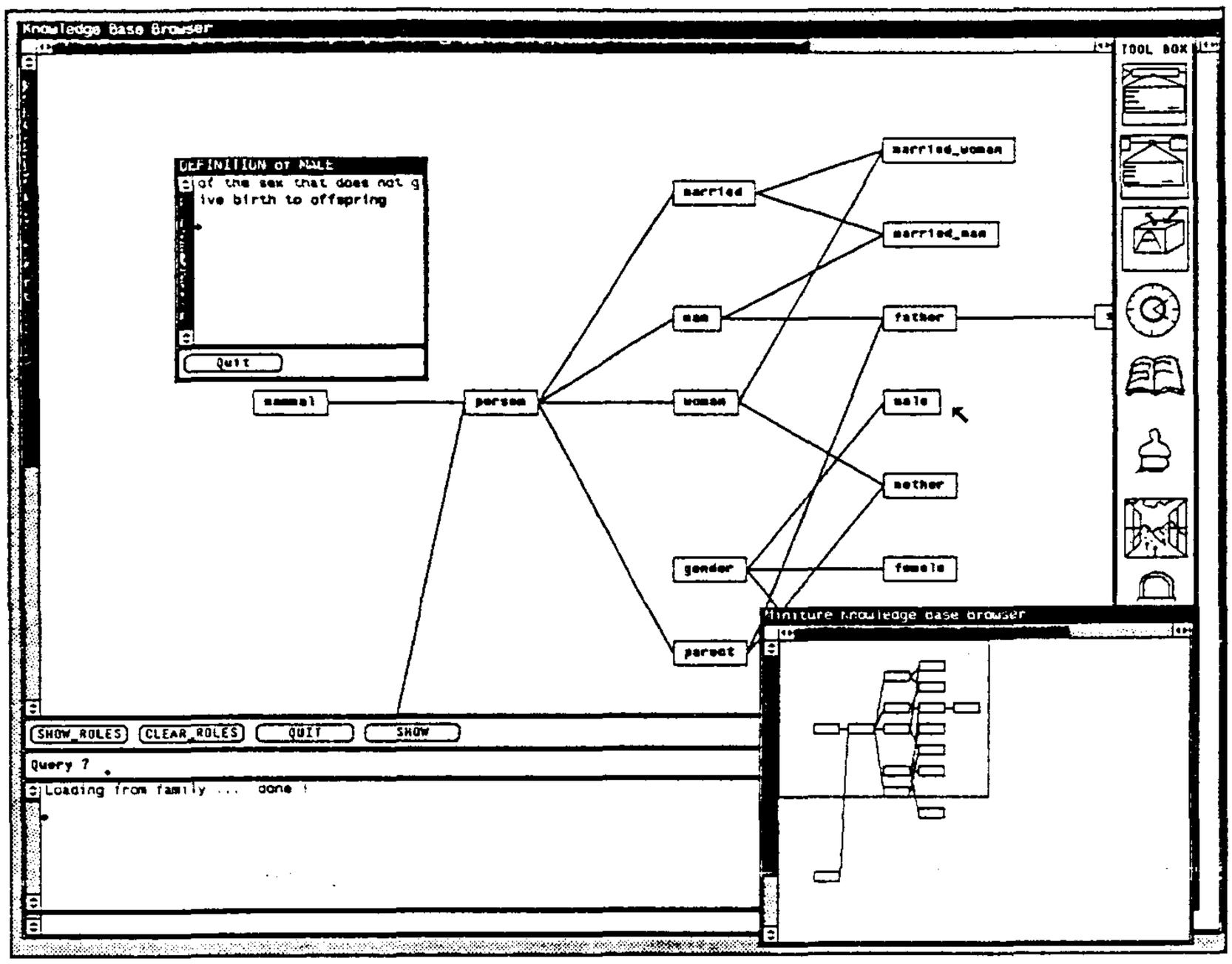
<그림 3.25>


Knowledge Base Browser
TOOL BOX
quit
person
married
man
woman
gender
parent
married_woman
married_man
father
male
mother
female
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

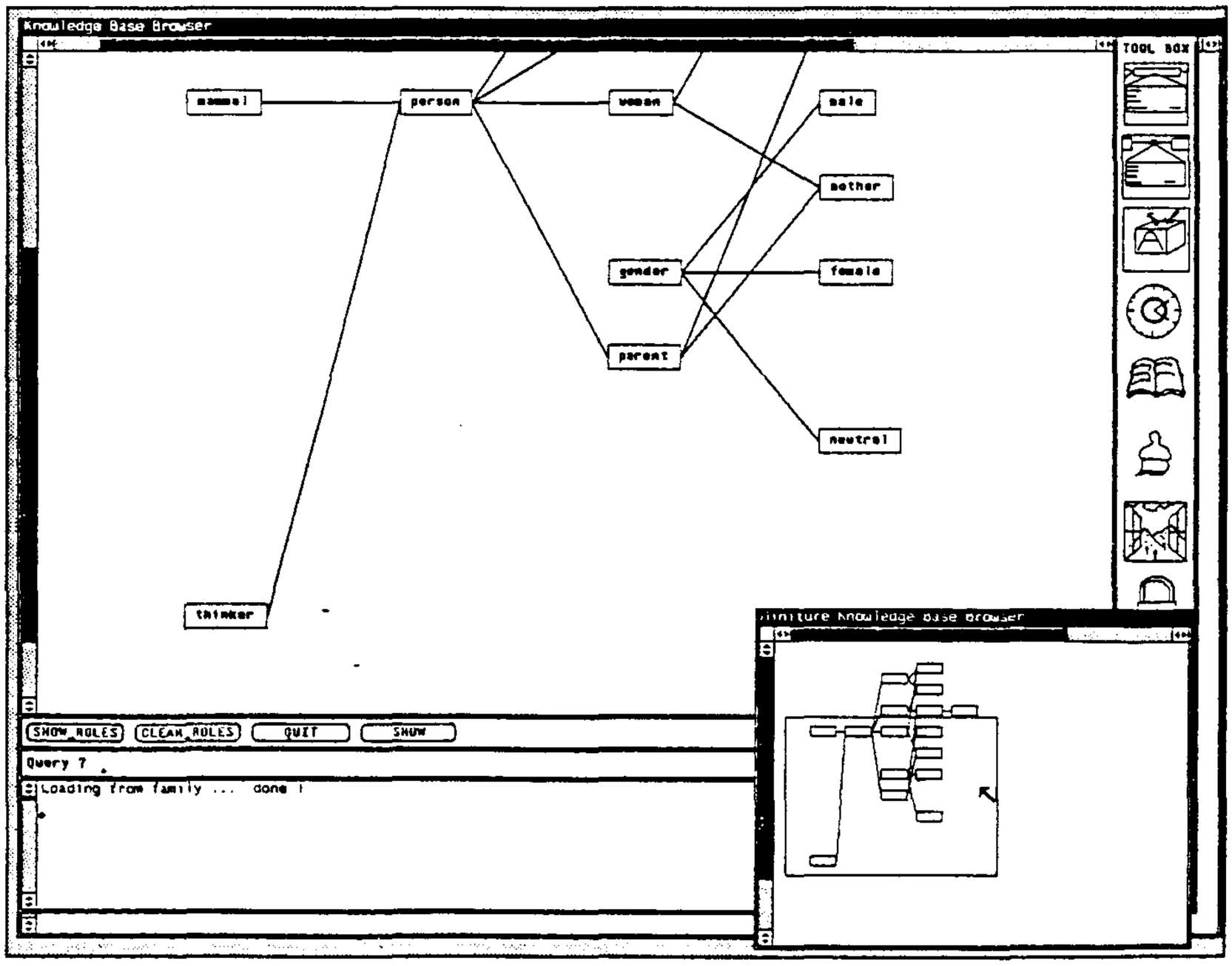
<그림 3.26>


Knowledge Base Browser
TOOL BOX
named
person
married
man
woman
gender
parent
married_woman
married_man
father
male
INSTANCE
DEFINITION
mother
female
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

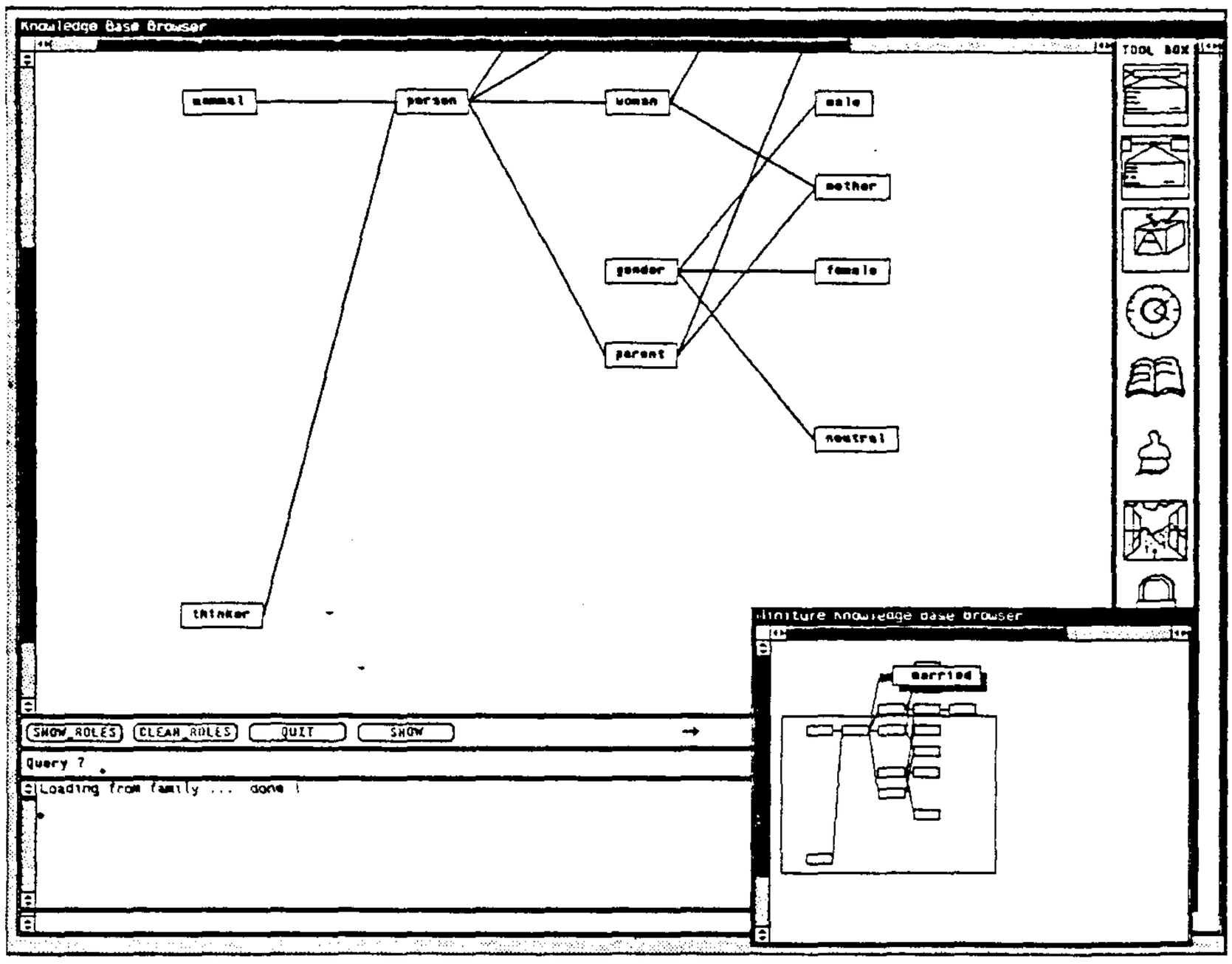
<그림 3.27>


Knowledge Base Browser
DEFINITION of MALE
of the sex that does not g
ive birth to offspring
Quit
TOOL BOX
married_woman
married_man
married
man
father
animal
person
woman
male
mother
gender
female
parent
Miniture Knowledge Base Browser
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

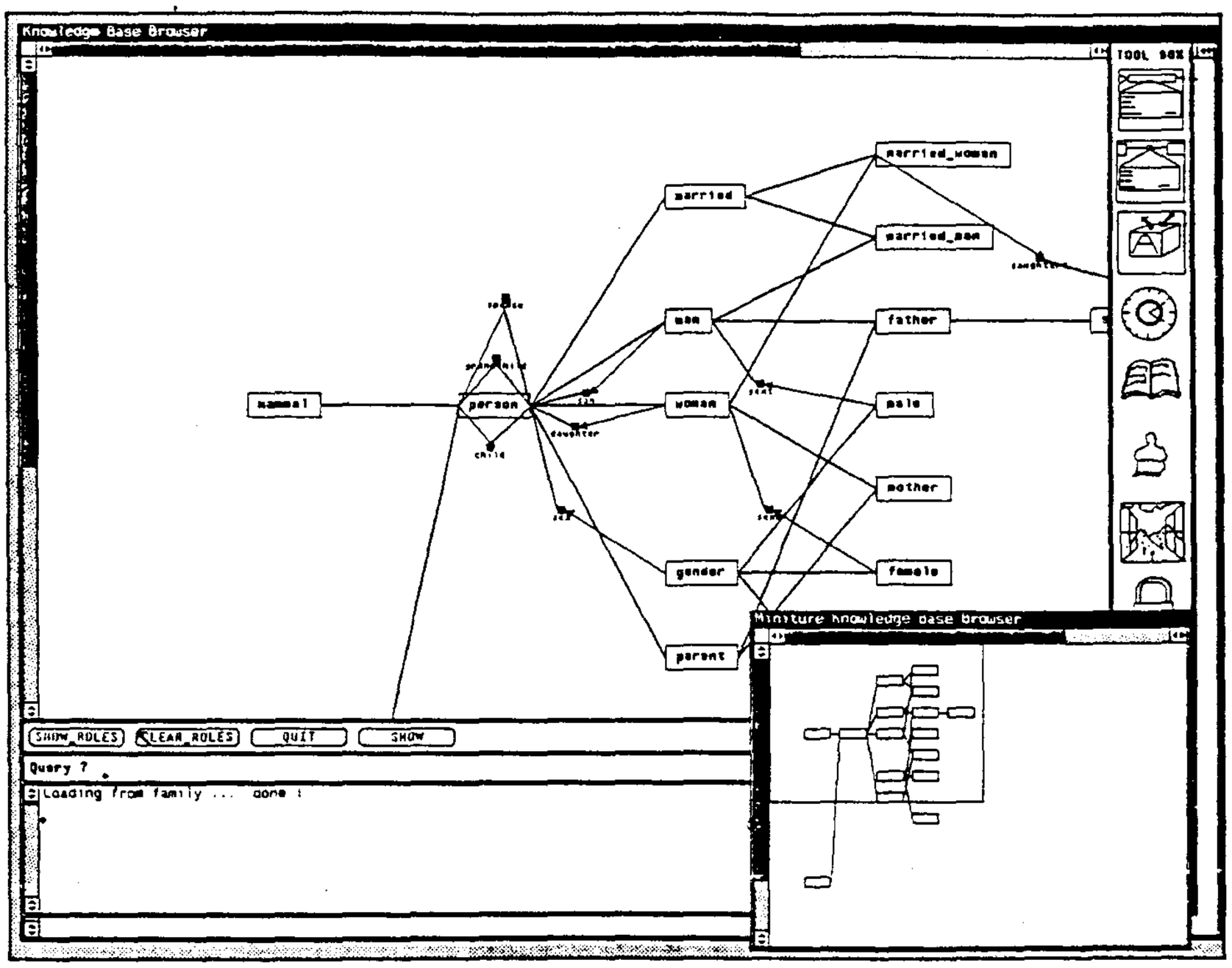
<그림 3.28>


Knowledge Base Browser
TOOL BOX
animal
person
woman
male
mother
gender
female
parent
neutral
thinker
Miniture Knowledge Base Browser
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !

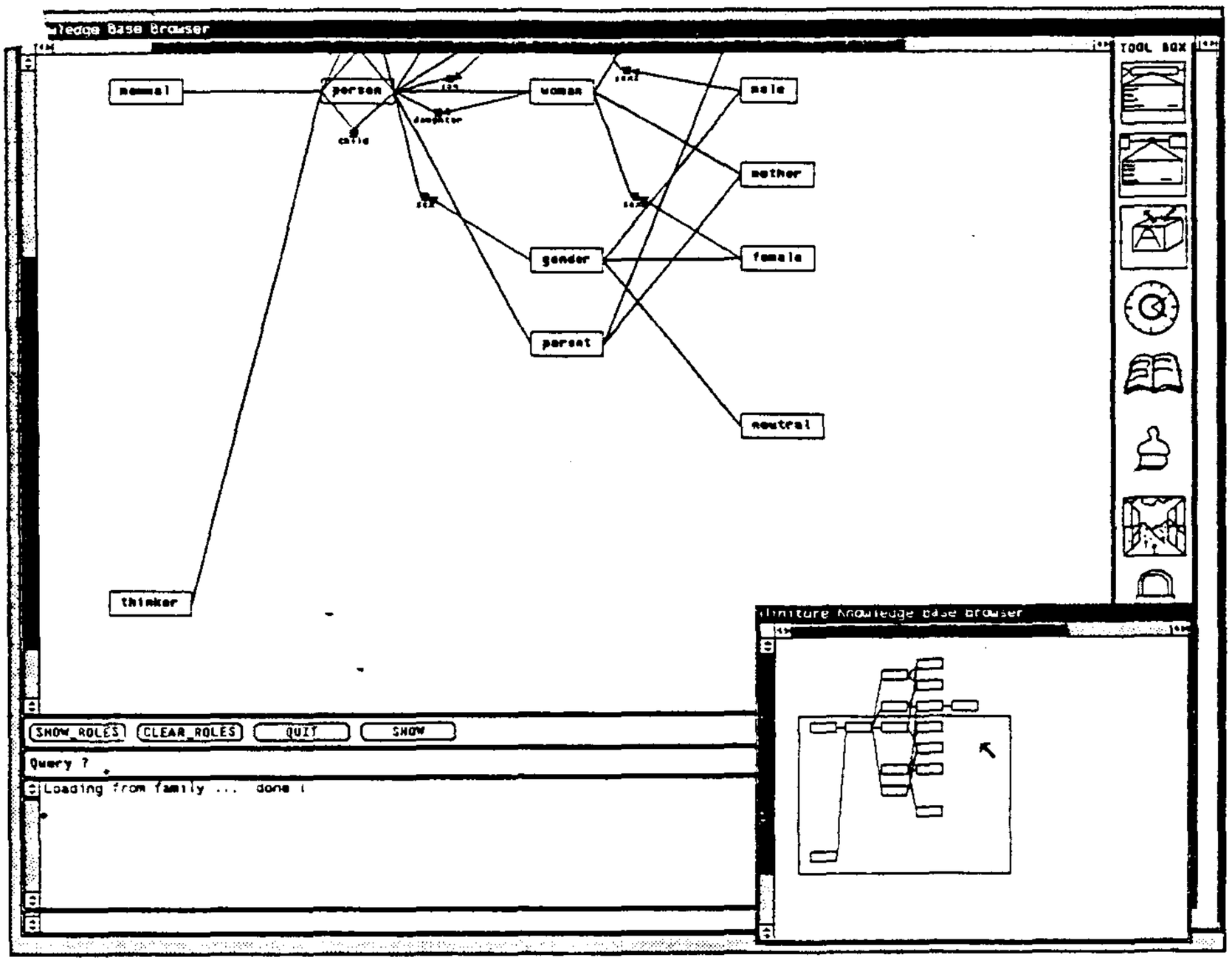
<그림 3.29>


Knowledge Base Browser
TOOL BOX
animal
person
woman
male
mother
gender
female
parent
neutral
thinker
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !
Miniture knowledge base browser
married

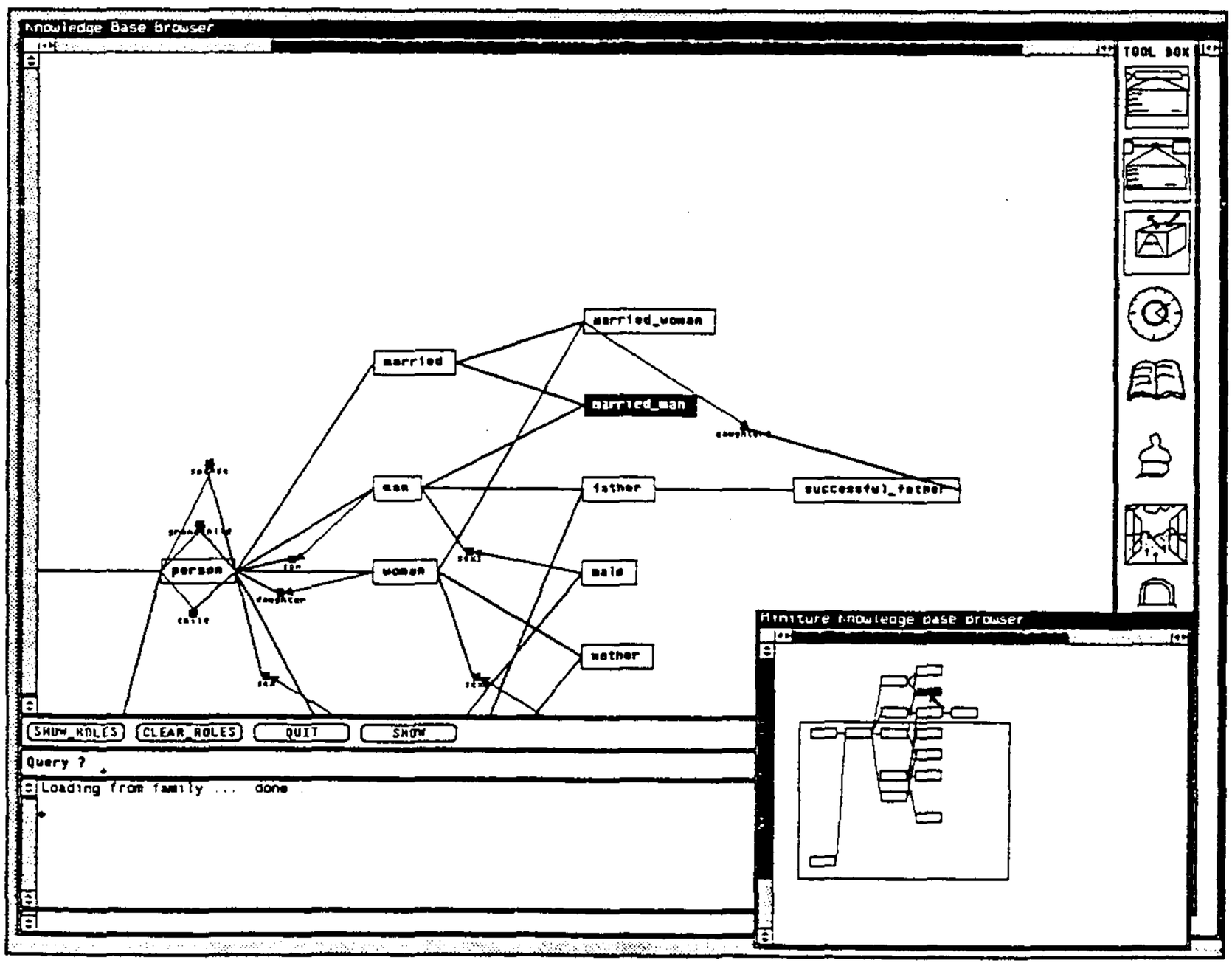
<그림 3.30>


Knowledge Base Browser
TOOL BOX
married_woman
married
married_man
man
father
animal
person
woman
male
mother
gender
female
parent
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done !
Miniture knowledge base browser

<그림 3.31>


ledge Base Browser
TOOL BOX
normal
person
woman
male
mother
female
gender
parent
neutral
thinker
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done
initure knowledge base browser

<그림 3.32>


Knowledge Base Browser
TOOL BOX
married_woman
married
married_man
man
father
successful_father
person
woman
male
mother
SHOW_ROLES
CLEAR_ROLES
QUIT
SHOW
Query ?
Loading from family ... done
Miniture knowledge base browser

<그림 3.33>

# 제 4 장 결론 및 앞으로의 연구 방향

본 연구에서는 Sphinx 시스템이라고 하는 혼성 지식 표현 시스템을 개발, 구현하였다. Sphinx는 여러 지식을 효과적으로 그리고 유기적으로 결합하여 처리할 수 있으며, 확장된 논리 프로그래밍 방법과 구조적 지식 표현 방법을 결합하여, 효율성과 시스템의 기능을 강화하였다. 특히 확장된 질의어는 한정사와 접속사를 분명히 사용할 수 있도록 하였으며, 대답의 치환을 통해 질의 응답 시스템의 기능을 할 수 있도록 하였다.

또한 NAF 규칙과 종속에 의한 되돌림 (recursion)을 통한 진리 유지 방식을 사용하여 저장 공간의 문제를 해결하였으며, 이에 바탕을 둔 설명 기능을 구현하였다.

실제로 사용할 수 있는 시스템으로 구축하기 위해 여러가지 프로그래밍 환경을 구축하였는데 이에는 지식베이스 편집기와 브라우저가 있으며, 이들을 사용하면 사용자가 지식베이스의 구조를 그래픽 형태로 파악할 수 있으며 보다 더 쉽게 지식베이스를 개발할 수 있다.

현재 대부분의 인공 지능 시스템에서 지식을 바탕으로 하고 있고 이를 표현하고 추론하는 것이 필요하기 때문에 Sphinx 시스템의 응용 분야는 매우 광범위하나 그 가운데서도 전문가 시스템이나 자연어 처리 분야에 한 모듈로 사용될 수 있다. 현재 Sphinx 시스템을 사용하여 참고 문헌에 관한 지식베이스를 구축하고 있으며 이를 확장하여 하나의 독립적인 질의-응답 시스템으로 개발하려고 한다. 또한 Sphinx 시스템의 기능을 확장하여 여기에 지식뿐만 아니라 믿음도 표현할 수 있도록 하고, 디폴트나 시간에 관한 지식을 첨가하거나, 이를 다중 agent를 위한 지식 표현 시스템으로 확장하는 것도 앞으로 수행해야 할 주요한 연구과제들이다.

# 참 고 문 헌


[Barr81] A. Barr, and E.A. Feigenbaum, The Handbook of Artificial Intelligence. Vol. I, Los Altos:William Kaufman, 1981.

[Brac83] R. J. Brachman, R.E.Fikes, and H.J. Levesque, "Krypton: A Functional Approach to Knowledge Representation," in IEEE Computer, Vol. 16, No. 10. pp. 67-73, 1983.

[Brac85a] R.J. Brachman, V. Pigman-Gilbert and H.J. Levesque, " An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of Krypton," in the Proc. of IJCAI-85, Los Angeles, California, 1985.

[Brac85b] R.J. Brachman, and J.G. Schmolze, "An Overview of KL-ONE Knowledge Representation System," Cognitive Science 9, pp.171-216, 1985.

[Clan85] W.J. Clancey, "Heuristic Classfication," *Artificial Intelligence*, Vol. 27, No.3, 1985, December.

[Clar78] K.L. Clark, "Negation as Failure," inCalliaire, H., and Minker, J. (Eds), *Logic and Data Bases,* New York:Plenum Press, pp. 293-322, 1978.

[Cutt86] Cutter Information Corp., *Expert Systems Strategies*,, Vol. 2, No. 7, pp. 7-16, 1986.

[Doyl78] J. Doyle, "Truth Maintenance Systems for Problem Solving," AI-TR-419, MIT AILAB, 1978, January.

[Han88] S. Han, D.W. Shin, Y. Kim, Y.P. Jun, S.R. Maeng, J.W. Cho, "A Logic Programming Approach to Hybrid Knowledge Representation," *Applied Artificial Intelligence*, Vol. 2, pp.93-127, 1988.

[Klee86] J. de Kleer, "An Asumption-based TMS," *Artificial Intelligence*, Vol. 28, No. 2. pp.127-162, 1986.

[Kunz84] J.C. Kunz, T.P. Kehler and M.D. Williams, "Application Development Using a Hybrid AI Development System," AI Magazine, Vol. 5, No.3, pp. 41-54, 1984. Fall.

[Lloy84] J.W. Lloyd, and R.W. Torpoor, "Making Prolog More Expressive," Journal of Logic Programming, Vol. 1, No.3, pp. 225-240, 1984, October.

[McAl82] D.A. McAllester, "Reasoning Utility Package User's Manual," AI Memo 667. MIT AI Lab, 1982.

[McAr85] G. McArthur, "APSN: A Hybrid System for Representing Belief and Knowledge," CSRI-169, CSRI, Univ. of Toronto, 1985, April.

[Nebe] B. Nebel, "Computational Complexity of Terminological Reasoning in BACK," KIT-REPORT 43, Technische Universitat Berlin, Berlin, Germany.

[Rich85] C. Rich, "The Layered Architecture of a System for Reasoning about Programs," in the Proc. of IJCAI-85, Los Angeles, California, 1985, August.

[Robi88] G. Robins, "Applications of the ISI Grapher", *Proceedings of the Artificial Intelligence and Advanced Computer Technology Conference*, Long Beach, California, 1988.

[Schm83] J.C. Schmolze, and D.J. Israel, "KL-ONE:Semantics and Classification," in Sidner, C.L. (Ed.), *Research in Knowledge Representation and Natural Language understanding --Annual Report, 1 September 1982-31 August 1983.* Bolt Bernaek and newman Report No. 5421, 1983.

[Stef86] M. Stefik, and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," AI Magazine, Vol. 6 No.4, 1986, Winter.

[Ster86] L. Sterling, and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*. Cambridge: The MIT Press, 1986.

[Stic83] M.E. Stickel, "Theory Resolution: Building in Nonequational Theories," in the Proc. of AAAI-83, Washington DC, pp.391-397, 1983, August.

[Vila84] M. Vilain, "KL-TWO, A Hybrid Knowledge Representation System," in Research in Knowledge Representation for NAtural Language Annual Report, BBN Report 5694, pp. 9-29, 1984, September.

[Vila85] M. Vilain, "The Restricted Language Architecture of a Hybrid Representating System," in th e Proc. of IJCAI-85, Los Angeles, California, 1985, August.