병렬 DEVS 시뮬레이션 환경(P-DEVSIM++)의 성능 평가

Performance Evaluation of a Parallel DEVS Simulation Environment P-DEVSIM++

성영락*, 김탁곤*, 박규호*
Yeong Rak Seong, Tag Gon Kim and Kyu Ho Park

Abstract

Zeigler's DEVS(Discrete Event Systems Specification) formalism supports formal specification of discrete event systems in a hierarchical, modular manner. Associated are hierarchical, distributed simulation algorithms, called abstract simulators, which interpret dynamics of DEVS models. This paper deals with performance evaluation of P-DEVSIM++, a parallel simulation environment which implements the DEVS formalism and associated simulation algorithms in a parallel environment. Performance simulator has been developed and used to experiment models of parallel simulation executions in different conditions. The experimental result shows that simulation time depends on both the number of processors in the parallel system and the communication overheads among such processors.

1. 서론

그 동안 이산사건 시스템을 모델링하고 시뮬레이션하는 것에 대한 연구가 진행되어 왔다. DEVS(Discrete Event System Specification)는 이산사건 시스템을 기술하는 형식론이다[9][10]. DEVS는 시스템을 작은 모듈들로 나누고 그것들을 계층적으로 구성해 나간다. 각 모듈들은 atomic 모델로 표현되며 그것들의 계층적 구성은 coupled 모델로 표현된다. DEVS 형식론에서는 또 DEVS로 모델링된 시스템의 시뮬레이션을 위하여 추상화 시뮬레이터(abstract simulator) 알고리즘을 제공한다. 추상화 시뮬레이터의 종류에는 simulator와 coordinator가 있으며 이것들은 각각 atomic 모델과 coupled 모델을 위한 시뮬레이터이다. 이때 모델들과 추상화 시뮬레이터들은 일대일 대응 관계를 가진다. 즉, 하나의 모델은 하나의 추상화 시뮬레이터와 항상 쌍을 이루게 된다. DEVS의 추상화 시뮬레이터는 그동안의 여러 연구에서 구현되었다. 그 중에서 DEVSIM++는 C++ 환경에서 구현된 DEVS 추상화 시뮬레이터이다[6][8].

P-DEVSIM++는 DEVSIM++의 시뮬레이션 속도를 향상시키기 위하여 DEVSIM++를 분산 환경에 이식한 것이다[1]. 그것을 위하여 두 클래스 Monitor와 ModelID

* 한국과학기술원, 전기 및 전자공학과, 컴퓨터 공학연구실

를 도입하였다. 그 중에서 Monitor 클래스는 분산 시뮬레이션을 위한 커널의 역할을 한다. 모니터는 일반적인 경우-성능 측정 시뮬레이터 환경같은 특별한 환경을 제외한 경우-분산 시스템의 각 노드에 하나씩 대응된다. 모니터의 역할은 초기화, 스케쥴링, 메시지 전송의 제어 등이 있다. 어떤 한 노드에 P-DEVSIM++가 다운로드되면 제일 먼저 모니터는 사용자가 정의한 입력 화일을 읽어서 그 노드에 대응될 DEVS 모델과 추상화 시뮬레이터를 생성하고 그것들의 데이타를 초기화한다. 이 과정이 끝난 뒤에 최상위 추상화 시뮬레이터인 root coordinator를 관리하는 모니터가 시뮬레이션 시작 메시지를 만들어서 root coordinator에게 전달하면 이때부터 시뮬레이션이 시작된다. 추상화 시뮬레이터는 항상 모니터로부터 메시지를 받아서 동작을 수행하므로, 모니터는 메시지의 흐름을 제어하여 추상화 시뮬레이터들이 일을 처리하는 것을 스케쥴링할 수 있다. 시뮬레이션이 진행되는 도중에 어떤 추상화 시뮬레이터에서 메시지가 발생되면 그것은 모니터에게 전달된다. 그러면 모니터는 그 메시지가 최종적으로 전달되어야 할 추상화 시뮬레이터가 어떤 노드에 대응되었는가를 판단한다. 만약 이때 추상화 시뮬레이터가 같은 노드에 대응되어 있다면 모니터는 그 메시지를 자신의 메시지큐에 집어 넣고 그렇지 않을 경우에는 그 추상화 시뮬레이터가 대응된 노드의 모니터에게로 전달한다. 그리고 분산 시뮬레이션을 위한 데이타는 클래스 ModelID에 의해 관리된다. 여기에서는 각 모델들의 우선순위, 해당 추상화 시뮬레이터들의 대응정보, 다음 내부사건 스케쥴 시간 등을 관리한다.

P-DEVSIM++는 DEVS 추상화 시뮬레이터의 내부사건과 외부사건을 병렬처리함으로써 시뮬레이션 속도를 향상시켰다[1]. 외부사건의 병렬처리는 Zeigler와 Zhang에 의해 제안된 계층적 시뮬레이션 알고리즘을 이용하였다[11]. 이 알고리즘은 보수적(conservative) 분산 시뮬레이션 알고리즘으로서, 외부사건의 처리는 다른 모델들의 수행에 영향을 주지 않기 때문에 보수적인 알고리즘을 사용하는 것이 적합하다. 그것에 비해 내부사건의 처리는 다른 추상화 시뮬레이터의 외부사건을 야기하므로 새로운 알고리즘 PAR_INT를 도입하였다. 이 알고리즘은 낙관적(optimistic) 분산 시뮬레이션 알고리즘인 Time Warp의 개념을 이용하였다. 그러나 일반적인 Time Warp 알고리즘과는 달리 PARINT는 rollback이 그것을 시작한 하나의 오브젝트-PDEVSIM++의 경우에는 추상화 시뮬레이터-내에서 국한되는 성질이 있어서 낙관적 시뮬레이션의 오버헤드를 줄인다.

본 연구에서는 P-DEVSIM++의 성능을 분석하고자 한다. 일반적으로 시스템이나 알고리즘의 성능의 분석을 위하여 널리 사용되는 방법으로 해석적 모델링, 실제 환경에서의 실험, 시뮬레이션 등이 있다[5]. 그중에서 해석적 모델링의 경우에는 비교적 쉽게 사용할 수 있으나 해석을 위해 도입된 가정이나 단순화 작업에 의하여 정확도가 떨어지는 단점이 있다. 그리고 실제 환경에서의 실험은 현실적이기는 하지만 일반적으로 실험 환경의 여러 변수들은 정확한 값을 알기도 어렵고 또 그 값들을 조정하기도 어렵기 때문에 많은 한계를 가진다. 거기에 비해 시뮬레이션은 모델링 과정에서의 추상화 정도에 따라 비교적 정확한 값을 구할 수 있고 또 실험 변수들을 용이하게 조정할 수 있다. 그러므로 본 연구에서는 시뮬레이션을 통하여 P-DEVSIM++의 성능을 분석하였다. 그것을 위해 P-DEVSIM++의 시뮬레이션 동작을 모델링하고 그것을 성능 측정 시뮬레이터로 구현하였다. 외부 사건만을 병렬처리하는 DEVS 추상화 시뮬레이터에 대해 성능 측정 시뮬레이터를 이용한 성능 평가는 [12] 등에서 연구된 바가 있다.

2 장에서는 본 연구의 이론적인 근거가 되는 DEVS 형식론의 모델과 추상화된 모니터에 대해 알아보겠다. 또 3 장에서는 P-DEVSIM++의 시뮬레이션 동작을 모델링하고 그것을 이용하여 성능 측정 시뮬레이터를 설계하겠다. 4 장에서는 설계된 성능 측정 시뮬레이터 환경에서 벤치마크 시스템 모델을 이용하여 P-DEVSIM++의 성능을 측정하였다. 5 장은 결론이다.

2. DEVS 형식론

DEVS 형식론은 대상 시스템을 모듈화된 여러 작은 모델들로 나누고 그것들을 계층적으로 구성하는데, 이는 DEVS 형식론이 대상 시스템의 동작과 구조를 분리하여 기술하기 때문이다. 그 기본이 되는 모델들을 atomic 모델이라고 하며 다음과 같이 표현한다[9][10].

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

여기서

X 입력 사건의 집합.
S 일련의 상태의 집합.
Y 출력 사건의 집합.

δ_{int} : S → S 내부 상태전이 함수.

δ_{ext} : Q × X → S 외부 상태전이 함수.

λ : S → Y 출력 함수.

ta : S → $R^{+}_{0,\infty}$ 시간 진행 함수.

여기서 Q는 M의 전체 상태를 나타내는 것으로서 일반적인 상태변수들에 의해 표현되는 모델의 상태에 시간을 포함한 상태를 의미는 것으로 다음과 같이 주어진다.

$$Q=(s,e) \mid \in S \text{ and } 0 \leqq e \leqq ta(s).$$

위에서 나타난 것과 같이 DEVS atomic 모델은 두 개의 상태전이 함수를 가진다. 그 중에서 내부 상태전이 함수 δ_{int}는 시간이 진행됨에 따라 일어나는 상태의 변화를 나타내기 위한 함수이고, 외부 상태전이 함수 δ_{ext}는 어떤 상태에서 외부로부터 입력을 받아서 그 상태가 바뀌는 것을 표현하는 함수이다. 그리고 출력 함수는 내부 상태전이가 일어날 때 수행되어서 외부로 출력을 발생하는 함수이다. 이때 발생한 출력은 다른 모델들의 입력이 되므로 한 atomic 모델의 내부 상태전이는 다른 atomic 모델들의 외부 상태전이를 야기하게 된다. 그러나 외부 상태전이에서는 출력이 발생되지 않으므로 다른 모델들에게 영향을 주지 않는다.

다음의 형태는 coupled 모델로 구성요소가 되는 몇 개의 모델들을 결합하여 새로운 모델을 만드는 것을 표현한다. 그런데 이 coupled 모델은 또한 다른 모델의 구성요소 모델이 될 수 있기 때문에 이것을 이용하여 복잡한 모델을 계층적으로 구성할 수 있게 된다[3][4]. Coupled 모델 DN은 다음과 같이 정의된다[9][10].

$$DN = \langle D, M_i, I_i, Z_{i,j}, \text{select} \rangle$$

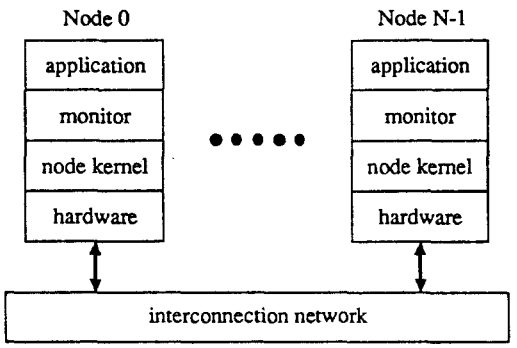
여기서,

D 구성요소들의 이름의 집합.
for each i in D
M_i D의 i번째 구성요소의 DEVS.
I_i i번째 모델의 influencee 모델들의 집합.
for each j in I_i
$Z_{i,j}$: $Y_i \rightarrow X_j$ · i번째 모델의 출력을 j번째 모델의 입력으로 연결하는 함수.
select : subsets of $D \rightarrow D$ 여러구성요소 모델들이 같은 시간에 스케줄을 원할때 그 중에서 하나를 고르는 함수.

여기서 select 함수는 일반적인 순차형 컴퓨터에서 한 시간에 여러 모델이 동시에 수행될 수 없으므로 그것들을 순서적으로 처리하는 역할을 한다.

DEVS 추상화 시뮬레이터에는 (*, t), (x, t), (y, t), (done, t_N)의 네 종류의 메시지가 존재한다. 내부사건 메시지 (*, t)는 내부 상태전이를 알리는 메시지로서 내부 상태전이가 일어날 시뮬레이션 시간이 되었음을 알려주는 역할을 한다. 즉 어떤 simulator가 이 메시지를 받으면 그 simulator는 자신이 관리하는 atomic 모델이 출력 함수와 내부 상태전이 함수를 수행하게 한다. 이때 출력 함수에 의해 발생되는 출력 메시지를 (y, t)로 표시한다. 그 (y, t) 메시지는 그 simulator의 상위 coordinator에게 전달되고 coordinator는 그것을 자신이 관리하는 coupled 모델의 coupling 정보를 이용하여 자신의 하위 simulator나 coordinator 혹은 자신의 상위 coordinator에게 전달한다. 위의 coupling 과정을 반복하여 결국 그 메시지는 다른 simulator의 입력으로 전달된다. 이때 그 입력 메시지는 그 simulator의 atomic모델의 외부사건을 야기하므로 그것을 외부사건 메시지라 하고 (x, t)로 표시한다. 즉 (y, t) 메시지는 coordinator들에 의해 (x, t) 메시지로 변환된다. 마지막으로 (done, t_N) 메시지는 다음 내부사건이 일어날 시간을 상위의 coupled 모델에게 알리기 위하여 사용된다. 어떤 atomic모델이 외부 상태전이 혹은 내부 상태전이를 수행하고 나면 상태가 바뀌므로 다음의 내부 상태전이가 일어나는 시간이 바뀐다. 그것을 상위의 coupled 모델들에게 전달하기 위해서 그 atomic 모델의 simulator는 (done, t_N) 메시지를 발생시킨다. 이때 tN은 그 atomic 모델의 시간 진행 함수에 의해 구한 것으로 스케줄된 다음 내부사건의 발생 시간이다. 그 메시지를 받은 coordinator의 coupled 모

델은 자신의 하위 모델들의 t_N 중에서 최소의 값으로 자신의 t_N 을 정하고 역시 그것을 상위의 coupled 모델에게로 전달한다. 결국 최상위 coordinator인 root coordinator는 전체 시스템의 atomic 모델의 내부사건들 중에서 가장 먼저 일어나야 할 내부사건의 시간 t_N을 알 수 있으므로, 시뮬레이션 시간을 그 시간까지 진행시키고 그것을 (*, t) 메시지로 하위의 추상화 시뮬레이터들에게 알려준다. 여기서 t는 새로 바뀌어진 시뮬레이션 시계의 시간이다. (*, t) 메시지를 받은 coordinator는 자신의 하위 추상화 시뮬레이터들 중에서 t_N이 (*, t)의 t와 같은 추상화 시뮬레이터에게 그 메시지를 전달한다. 이 과정을 반복하여 결국 그 메시지는 가장 먼저 내부사건을 수행해야 할 atomic 모델의 simulator에게 전달된다. 상세한 설명은 [3]를 참조하라.



〈그림 1〉 P-DEVSIM++의 계층구조

3. P-DEVSIM++의 성능 측정 시뮬레이터

이 장에서는 P-DEVSIM++의 성능 측정 시뮬레이터에 대해 살펴보겠다. P-DEVSIM++의 성능 측정 시뮬레이터는 P-DEVSIM++의 시뮬레이션 동작을 시뮬레이션하는 시뮬레이터로서 P-DEVSIM++의 성능을 측정할 수 있는 것 외에, 시뮬레이터의 여러 변수들을 변화시켜 가면서 시뮬레이션하면 특정 변수에 대한 성능의 변화추이도 관측할 수 있다.

그림 1은 P-DEVSIM++를 어떤 분산 시스템에서 수행할 때 그 시스템의 각 노드내에서의 제어 계층구조이다. 최상위 계층은 응용 계층으로 시뮬레이션할 대상 시스템의 DEVS 모델들과 추상화 시뮬레이터로 구성된다. 두번째 계층은 모니터 계층이다. 모니터의 역할은 1 장에서 설명하였다. 세번째 계층은 노드 커널 계층이다. 이 계층에서는 하드웨어 계층을 제어하고 모니터 계층이나 하드웨어 계층으로부터 전달된 통신 요구를 처리하는 계층이다. 네번째 계층인 하드웨어 계층은 실제 프로그램이 수행되고 통신이 이루어지는 물리적인 환경이다.

성능 측정 시뮬레이터를 구현하기 위해서는 P-DEVSIM++의 시뮬레이션 동작을 모델링해야 한다. 그런데 P-DEVSIM++는 분산 환경에서 동작하는 것이므로 분산 환경을 이루고 있는 노드의 특성[예를 들어, 다중통신(multicasting) 능력, 연결망 구조, 통신 전용 프로세서의 유무 등]이 P-DEVSIM++의 시뮬레이션 동작에 영향을 준다. 그림 1에서 상위의 두 계층은 P-DEVSIM++이 시뮬레이션 환경에 대해 독립적인 부분이고 하위의 두 계층은 환경에 직접적인 관련이 있는 부분이다.

3.1 절에서는 P-DEVSIM++의 시뮬레이션 환경에 대해 독립적인 부분을 모델링하고 3.2절에서는 3.1절의 내용에 시뮬레이션 환경과 관련된 부분을 더하여 P-DEVSIM++의 성능 측정 시뮬레이터를 설계하였다.

3.1 P-DEVSIM++의 모델링

이 절에서는 P-DEVSIM++의 시뮬레이션 동작 중에서 시뮬레이션 환경에 대해 독립적인 부분을 모델링하겠다. 이 부분은 그림 1에 나타난 계층 중에서 상위의 응용 계층과 모니터 계층이 해당된다. 그런데 P-DEVSIM++의 환경에 독립적인 시뮬레이션 동작을 모델링할 때 고려해야 할 사건은 어떤 메시지의 처리가 끝나는 사건과 메시지의 처리과정에서 출력이 발생되는 사건이다. 왜냐하면, 앞의 사건에 의해 우리는 다음 메시지를 처리하도록 할 수 있고, 뒤의 사건에 의해 새로운 메시지가 발생되는 것을 알 수 있기 때문이다. 그런데 그러한 사건들은 모두 응용 계층에서만 발생한다. 즉, 모니터 계층에서는 주어진 사건을 처리하기만 할 뿐이지 스스로 사건을 발생시키는 경우는[1] 없다. 그리고 응용 계층도 더 세분하면 DEVS 모

1) 초기화 과정에서 시작 메시지를 발생하는 것은 제외.

델들과 추상화 시뮬레이터들로 나뉘는데 그 중에서 DEVS의 모델들은 시뮬레이터에 의해 제어되는 수동적인 요소이므로 응용 계층과 모니터 계층에서 사건을 발생시키는 능동 소자는 추상화 시뮬레이터 뿐이다. 3.1.1 절에서는 simulator에 대해 3.1.2 절에서는 coordinator에 대해 살펴보겠다.

3.1.1 Simulator의 모델링

Simulator는 atomic 모델을 시뮬레이션하는 추상화 시뮬레이터이다. Simulator는 상위의 coordinator로 부터 (*, t)나 (x, t)메시지를 받아서 그것을 처리한 결과 발생한 (y, t)와 (done, t_N) 메시지를 그 상위 coordinator에게 보낸다. PAR_INT 알고리즘에 의하면 atomic 모델은 (x, t)메시지를 처리하기 위해서는 rollback해야 할 경우가 발생할 수 있다. 그러므로, 메시지가 전달되면 simulator는 그 메시지로 인하여 atomic 모델이 rollback을 해야 하는가를 판단하고 다음에 도착할 메시지에 의해 rollback해야 할 경우를 고려하여 atomic 모델의 현재 상태도 저장해 두어야 한다.

Simulator가 (*, t)메시지를 처리할 때 다음과 같은 과정을 거친다.

(가) Atomic 모델의 현재 상태를 저장한다. [T_0, T_1]

(나) Atomic 모델의 출력 함수를 호출한다. [T_1, T_2]

(다) 발생된 출력을 출력 메시지 (y, t)로 만들어 상위의 coordinator에게 전달한다. [T_2, T_4]

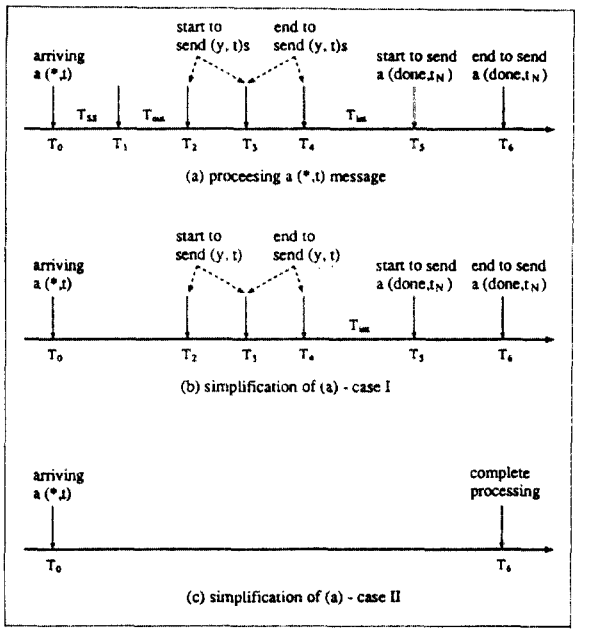
(라) Atomic 모델의 내부 상태전이 함수를 호출한다. [T_4, T_5]

(마) Atomic 모델의 시간 진행 함수를 호출하고 그 결과를 (done, tN) 메시지로 상위의 coordinator에게 전송한다. [T_5, T_6]

그림 2(a)는 위의 과정을 나타낸 것이다.

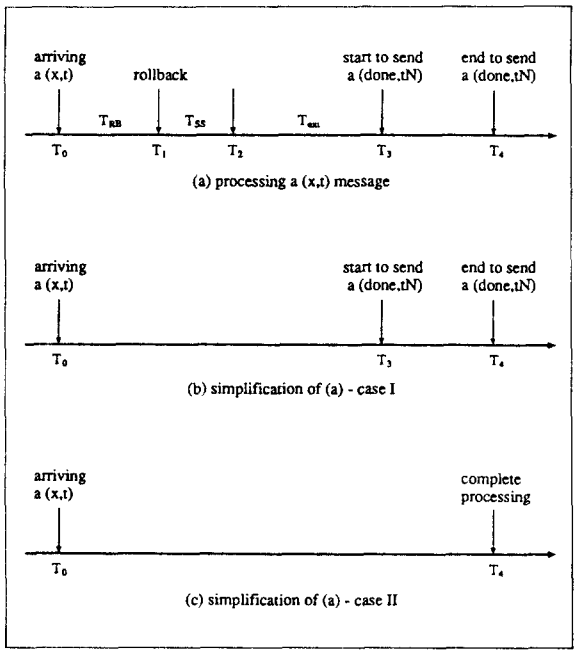
이제 위의 (*, t) 메시지 처리 모델을 단순화 해보자. 기본적인 접근 방법은 위의 여러 단계 중에서 처리하는 시간이 상수로 주어지는 단계는 다른 단계와 묶어서 한꺼번에 처리하는 것이다. 그것을 위하여 다음의 규칙 1을 정하였다.

규칙 1:한 노드내에서 일어나는 일련의 계산 과정에서 걸리는 시간은 어떤 상수 값으로 주어진다고 가정한다.

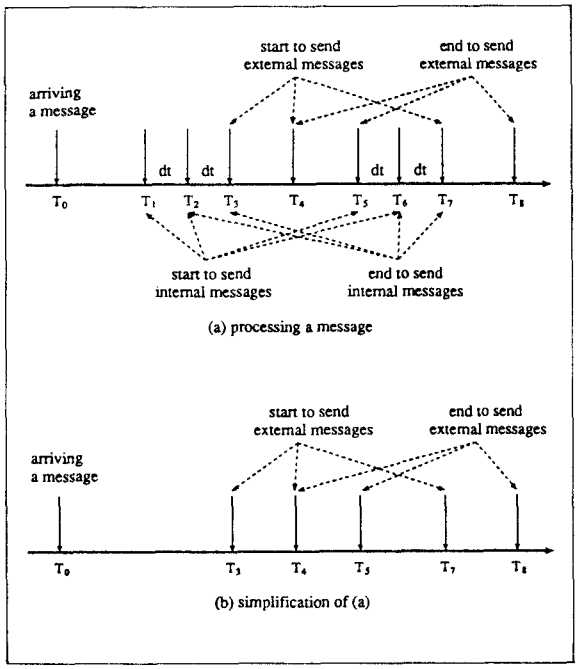


〈그림 2〉 Simulator의(*, t)메시지 처리

위의 (*, t)메시지 단계 중에서 (가), (나), (라) 단계는 모두 하나의 노드내에서 일어나는 일들이다. 그러므로 규칙 1에 의하여 우리는 이 단계를 수행하는 데에 걸리는 시간은 어떤 상수 값으로 정해져 있다고 가정할 수 있다. 거기에 비하여 (다), (마) 단계는 DEVS 모델들과 추상화 시뮬레이터가 어떻게 대응되었는가 하는 것에 따라 두가지의 경우가 발생한다. 즉 위의 단계들에서 발생된 메시지는 모두 모니터에게 전달되는데, 그러면 모니터는 그 메시지가 최종적으로 전달되어야 할 추상화 시뮬레이터가 어떤 노드에 대응되었는지를 판단하여 만약 같은 노드면 자신의 메시지큐에 집어넣고 다른 노드면 그 노드의 모니터에게 메시지를 전송한다. 그런데 simulator에서 발생되는 모든 메시지는 그 simulator의 상위 coordinator에게로 전달되므로 모니터는 항상 같은 결과를 갖는다. 그러므로 그 상위 coordinator가 만약 같은 노드에 대응되어 있다면 (다), (마) 단계의 처리 시간도 상수로 가정할 수 있고, 그렇지 않을 경우에는 그 단계들을 처리하는데 걸리는 시간은 시뮬레이션을 통하여 구해야 한다. 위의 관찰을 통하여 처리시간이 상수로 주어진 단계의 사건들은 성능 측정 시뮬레이션에서 중요한 사건들이 아니므로 그러한 단계들은 다른 단계들과 묶어서 처리 모델을 단순화시켰다.



〈그림 3〉 Simulator의(x, t)메시지 처리



〈그림 4〉 Coordinator의 메시지 처리

그림 2(b)와 (c)의 단순화된 (*, t) 메시지 처리모델을 나타낸 것이다.한편, simulator가 (x, t) 메시지를 처리할 때에는 다음과 같은 과정을 거친다.

(가) 현재 전달된 메시지의 처리를 위하여 rollback을 해야 할 지를 판단하고 필요하다면 그것을 수행한다.[T_0, T_1]

(나) Atomic 모델의 현재 상태를 저장한다.[T_1, T_2]

(다) Atomic 모델의 외부 상태전이 함수를 호출한다. [T_2, T_3]

(라) Atomic 모델의 시간 진행 함수를 호출하고 그 결과를 (done, tN) 메시지로 상위의 coordinator에게 전송한다. [T_3, T_4]

그림 3(a)은 위의 과정을 나타낸 것이다. (*, t) 메시지 처리 모델과 마찬가지로 규칙 1을 적용하여 그림 3(b), (c)과 같은 단순화된 (x, t) 메시지 처리 모델을 구할 수 있다.

3.1.2 Coordinator의 모델링

Coordinator는 coupled 모델을 위한 DEVS의 추상화 시뮬레이터이다. Coordinator는 상위의 coordinator로부터 (*, t), (x, t) 메시지를 받아서 하위의 추상화 시뮬레이터 (simulator 혹은 coordinator)에게로 전달하고 하위의 추상화 시뮬레이터로부터 (done, tN), (y, t) 메시지를 받아서 상위의 coordinator나 다른 하위의 추상화 시뮬레이터에게 전달한다. PAR_INT 알고리즘의 rollback을 위한 처리를 할 필요는 없다. 또 simulator와는 달리 coordinator는 하나 이상의 추상화된 시뮬레이터와 상호 메시지를 주고 받는다. 이때 그 대상 추상화 시뮬레이터들은 여러 노드에 나누어 대응될 수 있다.

Coordinator가 다루는 메시지는 여러 종류의 것이지만 그것들을 취급하는 방식은 매우 유사하다. 그래서 본 연구에서는 메시지의 종류에 따라 coordinator의 처리 모델을 분류하지 않았다. Coordinator가 사건 메시지를 받았을 때의 동작은 그림 4(a)와 같이 모델링되었다. T_0에서 coordinator는 상위나 하위의 추상화 시뮤레이터로부터 메시지가 전달되어야 할 추상화 시뮬레이터는 입력 메시지의 종류와 그 coordinator에 대응되는 coupled 모델의 정보에 의존한다. 같은 노드에 대응된 추상화 시뮬레이터들간의 메시지 전송에 걸리는 시간은 상수값으로 가정할 수 있으므로 앞의 처리 모델을 그림 4(b)처럼 단순화할 수 있다.

3.2 성능측정 시뮬레이터의 설계

이 절에서는 P-DEVSIM++의 시뮬레이션 환경에 관련된 부분을 분석하여 성능 측정 시뮬레이터를 개발한다. 그림 5는 P-DEVSIM++의 성능 측정 시뮬레이터를 수행 중인 노드의 제어구조이다. 그림 1의 P-DEVSIM++의 구조가 P-DEVSIM++의 응용 계층에 다시 반복되어 나타난 형태이다. 그리고 노드 커널과 하드웨어 계층은 새로운 계층-성능 측정 시뮬레이터 계층-으로 모델링되었다. 이 계층은 클래스 Psim으로 구현되었다. 클래스 Psim에 대한 상세한 내용은 3.2.1절에서 살펴보겠다.

그림 5를 보면 한 노드에 두 개의 모니터 계층이 있음을 알 수 있다. 그 각각의 계층에는 모니터가 하나씩 있다. 본 연구에서는 하위의 모니터는 커널 모니터라 하고, 상위의 모니터는 사용자 모니터라고 표현하겠다. 커널 모니터는 성능 측정 시뮬레이터 계층의 DEVS 모델들과 추상화 시뮬레이터를 제어하고, 사용자 모니터는 실제의 응용 모델과 추상화 시뮬레이터를 제어한다.

시뮬레이션 환경에 관련된 P-DEVSIM++의 동작을 모델링하기 위하여 본 연구에서는 다음과 같은 분산 환경을 가정하였다.

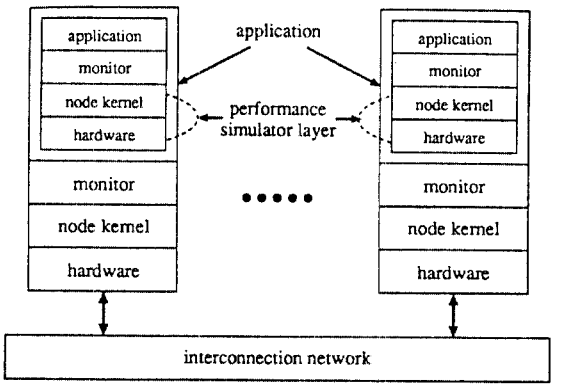
(가) 노드들은 하이퍼 큐브의 형태로 연결되어 있다.

(나) 노드에는 통신 전용 프로세서가 없다. 그러므로 통신과 계산(computation)이 동시에 수행될 수 없다.

(다) 노드간의 통신 시간은 상수이다. 일반적으로 노드간의 통신 시간은 $T_{comm} = T_{set\text{-}up} + T_{transmit}$로 주어진다. $T_{set\text{-}up}$은 통신채널의 여러 장치들의 레지스터를 맞추는데 걸리는 시간으로 메시지의 길이와는 무관하며, $T_{transmit}$는 앞의 채널로 데이타를 전송하는데 걸리는 시간으로 메시지의 길이에 따라 변하는 값이다. 본 연구에서는 메시지의 길이는 크게 변하지 않으며 또 $T_{set\text{-}up}$이 비교적 큰 값으로 가정하였다. 그러므로 노드간의 통신 시간 T_{comm}을 상수로 가정할 수 있다.

(라) 노드는 다중통신(multicasting) 능력이 없다. 노드는 동시에 여러 채널을 제어할 수 없는 것으로 가정하였다. 그러므로 하나의 메시지를 여러 노드에게 전송할 때에는 각각에 대하여 메시지 전송을 반복해야 한다.

(마) 노드 프로세서가 계산 작업을 처리하는 도중에 외부로부터 통신 요구가 발생하면 노드 프로세서는 즉시 자신의 일을 중단하고 통신 요구에 의한 인터럽트를 처리해야 한다.



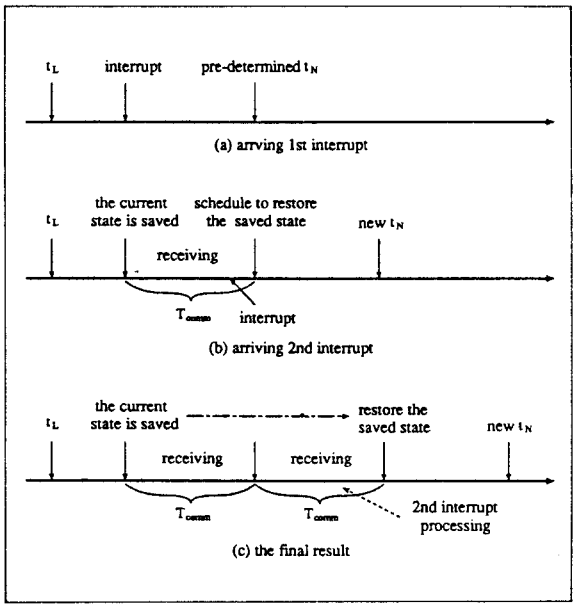
〈그림 5〉 P-DEVSIM++의 성능 측정 시뮬레이터의 계층구조

(바) 노드가 통신 작업을 처리하는 도중에 발생된 다른 통신 요구에 의한 인터럽트의 처리는 작업이 끝날 때까지 지연된다.

(사) 같은 시간에 한 노드에 여러 통신 요구가 생기면 그것을 발생한 노드들의 NID(노드 ID)를 비교하여 전체 시스템이 고착상태(deallock)에 빠지는 것을 방지한다. 예를 들어 노드 A가 노드 C로 메시지를 보내고자 하는 것과 동시에 노드 B가 노드 A에게 메시지를 보내고자 하면, 노드 A와 B의 NID를 비교하여 NID가 더 큰 노드의 통신 요구부터 먼저 처리한다.

그림 6은 노드 프로세서가 인터럽트를 처리하는 것에 대한 예이다. 어떤 노드 프로세서가 계산 작업을 처리하는 도중에 발생된 첫번째 인터럽트에 의해 현재 상태를 저장하고 인터럽트의 처리를 시작한다. 그리고 첫번째 인터럽트를 처리하는 도중에 두번째 인터럽트가 발생되어 두번째 인터럽트의 처리는 지연된다. 시간이 흘러 첫번째 인터럽트의 처리가 끝나면 두번째 인터럽트가 처리된다. 마침내 모든 인터럽트에 대한 처리가 끝나면 노드 프로세서는 저장된 상태를 복구하여 수행을 계속한다.

3.2.1 절에서는 클래스 Psim의 내부 구조를 살펴보고 3.2.2 절과 3.2.3절에서는 Psim의 내부사건과 외부사건 처리에 대해 알아보겠다.



〈그림 6〉 인터럽트의 처리

3.2.1 클래스 Psim

그림 7은 성능 측정 시뮬레이터 클래스 Psim을 나타낸 것이다. *monitor*는 Psim에 의해 제어되는 사용자 모니터를 가리킨다. 두 노드간의 통신 시간이 무시할 수 없는 값이며 노드 프로세서가 두 채널에서 동시에 메시지를 수신할 수 없으므로, 통신을 모델링하기 위하여 두 개의 자료구조를 사용하였다. 그중에서 *kInputQ*는 아직 처리되지 않은 인터럽트들을 보관하기 위한 자료구조이고, *uInputQ*는 인터럽트의 처리가 끝난 후에 데이타를 보관하기 위한 자료구조이다. 어떤 시간에 노드 프로세서가 외부로부터 통신 인터럽트를 받게 되면, Psim은 그것을 *kInputQ*에 저장한다. 그 후 통신 시간 T_{comm}이 지나면 Psim은 그 인터럽트의 처리를 끝마치게 되며, 이때 *kInputQ*에서 그 인터럽트에 대한 내용을 제거하고 수신한 데이타를 *uInputQ*에 저장한다. 이 데이타는 *monitor*가 통신 채널로부터 메시지를 읽도록 요청할 때—*removeMsg( )*를 호출하여—*uInputQ*에서 제거되어 *monitor*에게 전달된다. *eventQ*는 *monitor*가 발생한 사건들을 저장하는 자료구조이다. 사건에는

variables	functions	
monitor	addEvent( )	psimExt( )
kInputQ	removeMesg( )	psimOutput( )
uInputQ	psimInit( )	psimTa ( )
stateQ	psimInt( )	
eventQ		

〈그림 7〉 클래스 Psim

두 종류가 있다. 첫째 출력 사건은 *monitor*가 출력 메세지를 발생한 경우이고, 둘째 종료 사건은 *monitor*가 어떤 한 메시지의 처리를 끝냈을 때이다[2]. *monitor*는 Psim의 *addEvent( )*를 호출하여 발생된 사건들을 *eventQ*에 저장한다. *stateQ*는 계산 작업 중에 인터럽트가 발생할 때 현재 상태를 저장하기 위한 자료구조이다. *psimInit( ), psimInt( ), psimExt( ), psimOutput( ), psimTa( )*등은 Psim의 초기화 함수, 내부 상태전이 함수, 외부 상태전이 함수, 출력 함수, 시간 진행 함수이다.

두 노드간의 통신을 시뮬레이션하기 위하여 두 종류의 메시지를 사용하였다. 첫째 실제 메시지(real message)는 실제 데이타를 전송하기 위한 메시지고, 둘째 확인 메시지(acknowledgement message)는 두 노드가 통신을 끝마칠 때 동기를 맞추기 위한 메시지이다. 실제 메시지는 데이타를 전송하는 노드에서 발생되며, 확인 메시지는 데이타를 수신하는 노드에서 발생된다. 그러므로 실제 메시지는 통신의 시작을 의미하고 확인 메시지는 통신이 끝남을 의미한다.

Psim은 네 가지의 상태를 가진다. 첫째, COMPUTE 상태는 노드는 어떤 계산 작업을 수행 중에 있음을 의미한다. 그러므로 Psim은 내부사건이 일어나기를 기다려야 한다. 이때 그 내부사건이 일어나는 시간은 Psim의 시간 진행 함수에 의해 구해진다. 둘째, WAITING상태는 현재 노드내에는 처리해야 할 메세지가 없으므로 통신채널을 폴링(polling)하는 중임을 의미한다. 그러므로 이 때의 Psim의 t_N은 ∞이다. 세째, SENDMSG상태는 데이타를 전송하고 있는 상태이다. 이 상태에서는 데이타를 수신하는 노드에서 확인 메시지를 보낼 때까지 기다려야 하므로 t_N은 ∞이다. 마지막으로 RECVMSG상태는 노드는 현재 데이타를 수신하고 있음을 나타내는 것이다. T_{comm}만큼의 시

2) 실제로는 이 사건들은 monitor에 의해 제어되는 추상화 시뮬레이터들에 의해 발생된 것이다.

간이 지난 후에 이 노드는 실제 메시지를 전송한 노드에게 확인 메세지를 보내야 한다.

3.2.2 외부사건의 처리

그림 8은 Psim의 외부사건 처리를 나타낸 것이다. 확인 메시지를 받았을 때의 Psim의 상태는 인과 관계에 의해 SENDMSG이다. 메시지를 전송하고 있는 동안에 Psim은 외부의 통신 인터럽트에 대한 처리를 지연시켜 왔으므로, 확인 메시지를 받은 이후에는 *kInputQ*에 처리가 지연된 인터럽트가 있는지를 확인해야 한다. 만약 그러한 인터럽트가 있다면 그것을 처리하고—이때, Psim의 상태는 RECVMSG상태로 된다—, 그렇지 않을 경우 *stateQ*에 저장된 상태를 복구해서 그 일의 처리를 계속해야 한다. 그리고 만약 저장된 상태가 없다면—메시지를 받기 전의 상태가 WAITING인 경우—Psim은 *eventQ*에 전달된다. 종료

```
if the arrived message is an acknowledgement message then
    assert that the current state is SENDMSG;
    if kInputQ is not empty then
        state := RECVMSG;
        ta := Tcomm;
    else if a saved state exists
        restore the saved state and 'ta';
    else
        if there is no event in eventQ then
            call the user level monitor;
        end if
        state := COMPUTE;
        calculate the next event time and set 'ta';
    end if
else // the arrived message is a real message
    switch the current state
        case SENDMSG or RECVMSG.
            // interrupts are disabled
            // the message is added into the end of kInputQ
            add the message into kInputQ;
        case WAITING:
            state := RECVMSG;
            ta := Tcomm;
        case COMPUTE:
            // interrupt handing
            // save the current state and serve the request
            save the current state and left execution time;
            state := RECVMSG;
            ta := Tcomm;
        end switch
end if
```

〈그림 8〉 Psim의 외부사건 처리

```
switch
    case RECVMSG:
        // Eventually, this node finishes receiving a message
        remove the first element of kInputQ;
        insert the removed element into uInputQ;
        if kInputQ is not empty then
            // there are some suspended interrupt requests
            ta := Tcomm;
        else if a saved state exists
             restore the saved state;
        else
          call the user level monitor;
              // the user level monitor processes the received message
              // which makes some events of Psim
          determine ta;
        end if
    case COMPUTE:
        if there is no event in eventQ then
            call the user level monitor;
        end if
        calculate the next event time and set 'ta';
    case WAITING or SENDMSG:
        // not applicable
        terminate;
end switch
```

〈그림 9〉 Psim의 내부사건 처리

사건의 발생을 끝으로 다시 Psim에게 제어가 넘어가면, Psim은 *eventQ*에 처리해야 할 사건이 저장되어 있는지를 판단한다. 만약 그러한 사건이 있으면 그 사건에 의하여 다음 내부사건 시간이 정해지고 Psim은 COMPUTE상태가 되며, 그렇지 않을 경우에는 Psim은 WAITNG 상태이다.

이제 Psim의 실제 메시지 처리에 대해 살펴보자. 인터럽트의 처리가 지연되는 상태인 SENDMSG나 RECVMSG 상태에서 실제 메시지가 전달되면 그것에 의한 인터럽트의 처리도 지연된다. 그 인터럽트들은 앞의 확인 메시지 처리과정에서 언급한 바와 같이 우선 *kInputQ*에 저장된 뒤에 나중에 차례대로 처리된다. 이때 상태의 변화는 없다. 그리고 그밖의 상태에서 실제 메시지가 전달되면 Psim의 상태는 RECVMSG로 바뀌고 인터럽트는 즉각 처리된다. 특히 Psim이 계산 작업을 진행 중이었다면—COMPUTE상태—인터럽트를 처리하기 전에 현재 상태와 남은 처리 시간은 *stateQ*에 저장된다.

3.2.3 내부사건의 처리

그림 9는 Psim의 내부사건 처리 과정이다. Psim이 내부사건을 처리하는 경우는 RECVMSG나 COMPUTE상태의 경우이다. RECVMSG상태에서 내부사건이 일어나는 경우는 데이타의 수신이 끝났음을 의미하는 것으로 그 데이타를 송신한 노드에게 확인 메시지를 보내야 한다. 그리고 방금 수행을 마친 인터럽트의 정보를 *kInputQ*에서 제거하고 수신한 데이타를 *uInputQ*에 저장한다. 그 데이타는

나중에 사용자 모니터에게 전달될 것이다. 그 과정이 끝나면 *kInput()*에 다른 지연된 인터럽트가 확인해야 한다. 만약 그런 인터럽트가 존재하면 Psim은 그것을 계속해서 처리하므로 상태는 바뀌지 않는다. 더 이상 처리해야 할 인터럽트가 없다면 통신을 하기전에 *state()*에 저장해 둔 상태가 있는지를 조사한다. 만약 저장된 상태가 있다면 -이전 상태가 COMPUTE일 경우- 그것을 복구하고 수행을 계속한다. 그리고 그렇지 않다면--이전 상태가 WAITING일 경우- 사용자 모니터를 호출한다. 이때 사용자 모니터에 의해 새로운 사건이 발생하면 COMPUTE상태가 되고, 그렇지 않을 경우에는 WAITING 상태가 된다.

COMPUTE 상태에서 내부사건이 일어나는 경우는 Psim이 현재 수행 중인 계산 작업을 끝마쳤다는 것이다. 그러므로 상태의 변화없이 다음 작업의 처리를 스케줄하면 된다.

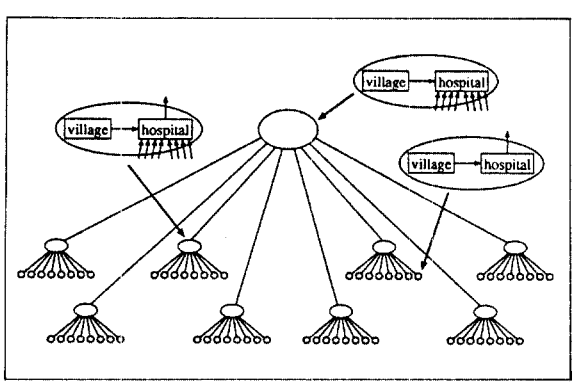
4. 실험

이 장에서는 성능 측정 시뮬레이터를 이용하여 PARINT알고리즘의 성능을 여러 실험 매개변수들을 바꾸어 가며 측정하였다. 이 실험에서 사용된 벤치마크 시스템은 Columbian Health Care System(CHCS)으로서 이산사건 시스템의 성능 평가를 위하여 널리 사용되었던 시스템이다[7][2]. 그러나 이전의 연구들은 형식론에 근거하지 않은 것에 비하여 본 연구에서는 DEVS 형식론에 근거하고 있다.

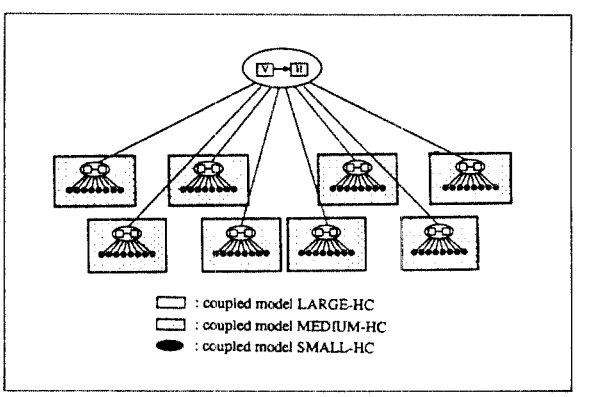
〈그림 10〉에 나타난 바와 같이 시뮬레이션 모델은 마을(village)과 병원(hospital)들로 구성된 3 단계 보건의료 시스템이다. 각각의 마을은 모두 지정 병원이 하나씩 있다. 또 병원들은 계층구조로 편재되어 만약 하위의 병원에서 치료할 수 없는 질병은 상위의 병원에서 치료하게 되어 있다. 그리고 각 병원에 대한 상위 병원은 이미 지정되어 있다. 어떤 마을에서 환자가 발생하면 우선 그 마을의 지정 병원으로 가게 된다. 각 병원에는 한 명 이상의 의사들이 있어서 도착한 환자들을 진료하게 된다. 이때 한 환자는 한 의사로부터 진료를 받게 된다. 만약 어떤 환자가 병원에 갔을 때 그 병원의 의사들이 모두 다른 환자들을 진료하느라고 바쁠 경우에는 그 환자는 자신의 차례를 기다려야 한다. 의사는 환자의 증상을 진찰한 후, 만약 그 질병이 그 병원에서 치료할 수 있다고 판단될 경우에는 그 병원에서 치료를 하고, 치료할 수 없을 경우에는 지정된 상위의 병원으로 환자를 보내게 된다. 그리고 최상위의 병원에서는 언제난 환자를 치료할 수 있는 것으로 한다.

이 모델에서 환자들은 외부사건 메시지로 표현된다. 즉, 마을에서 환자가 발생하여 병원으로 가는 것을 village 모델에서 내부사건을 수행하여 발생된 메시지가 hospital 모델의 입력으로 전달되는 것으로 모델링되었다. 각 village 모델은 독립적으로 정해진 수 만큼의 환자를 발생하는 것으로 모델링되었다. 각 hospital 모델은 메시지큐와 주어진 수 만큼의 의사들로 구성된다. 본 실험에서는 각 계층의 hospital 모델에는 16/4/1명씩의 의사가 있고 환자를 치료할 확률은 100%, 90%, 90%인 것으로 모델링하였다.

그림 11에는 CHCS 시스템 모델을 계층적으로 구성한 것을 나타내었다. 가장 상위의 coupled 모델은 LARGE-



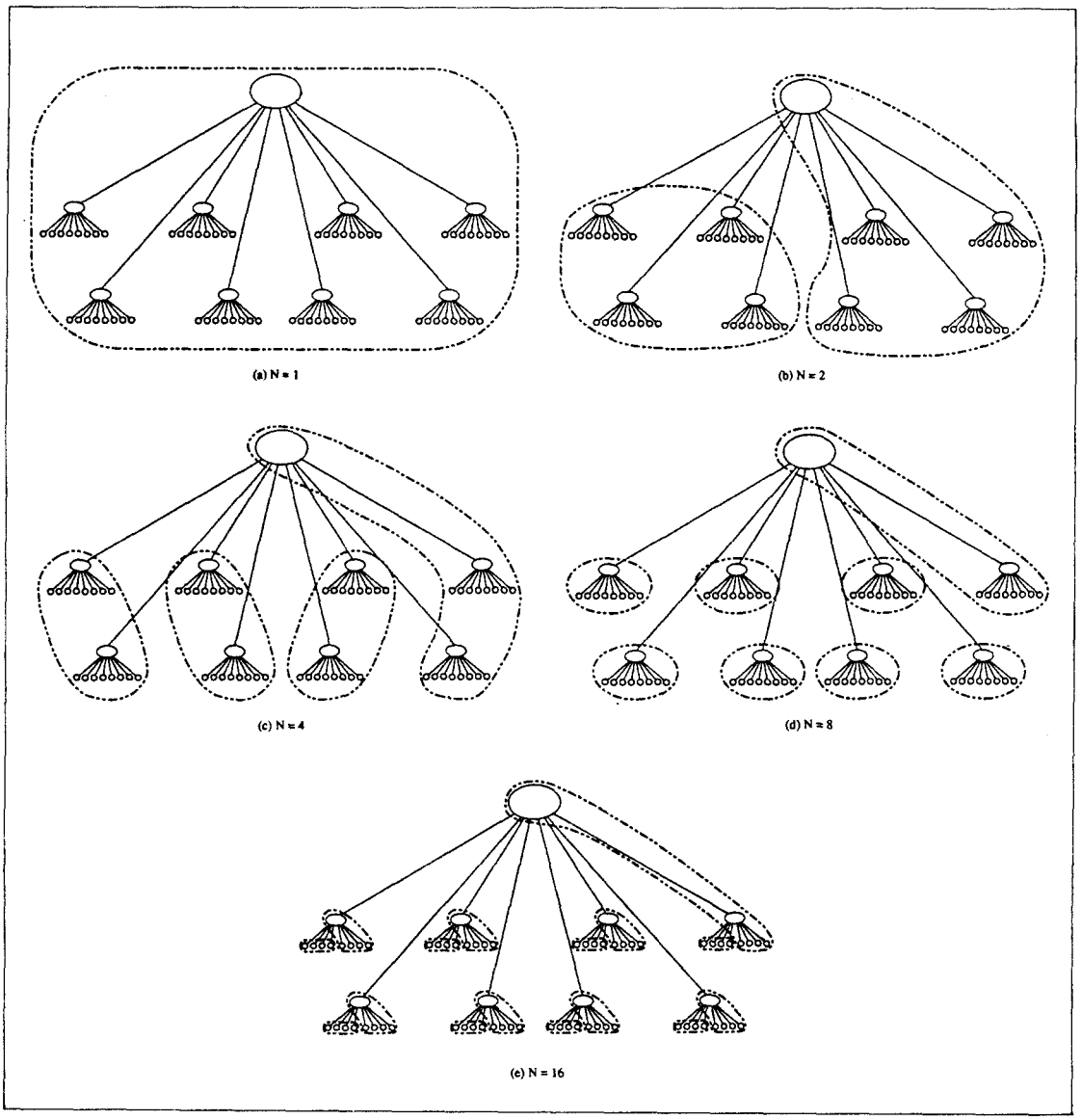
〈그림10〉 Columbian Health Care System



〈그림 11〉 CHCS의 계층적 구성

HC이며, 그것은 8개의 중간 계층 coupled 모델 MEDIUM-HC, 병원, 그리고 마을로 구성된다. MEDIUM-HC도 역시 같은 방법으로 8개의 하위 계층 coupled 모델 SMALL-HC, 병원, 그리고 마을로 구성된다. SMALL-HC는 병원과 마을로만 구성된다.

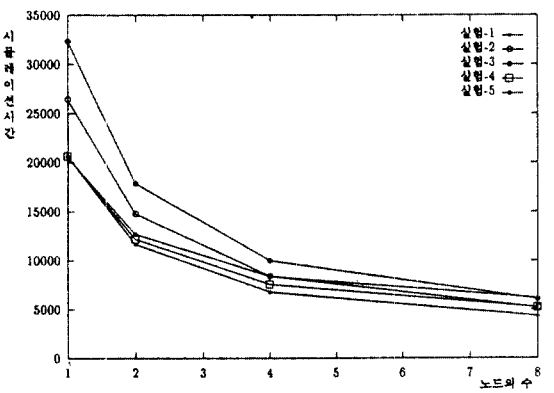
주어진 시스템 모델을 이용하여 시뮬레이션 사용하는 노드의 수를 변화시켜 가며 P-DEVSIM++의 성능을 측정하였다. 그림 12는 노드의 수가 1, 2, 4, 8, 16개일 때의 모델들의 대응을 나타낸 것이다. 통신 시간이 P-DEVSIM++의 성능에 미치는 영향을 실험하기 위하여 노드간의 통신 시간 T_{comm}, 노드내에서의 통신(메시지 전송) 시간 T_{MP}, village 모델과 hospital 모델의 내부·외부사건 처리 시간(계산 시간) T_{int} · T_{ext}를 표 1처럼 변화시켜 가면서 실험을 반복하였다.



〈그림 12〉 CHCS의 매핑

〈표 1〉 실험 변수

변 수	정 의	실험1	실험2	실험3	실험4	실험5
T_{comm}	노드간의 통신시간	1	1	1	2	3
T_{MP}	노드내에서의 통신시간	0	0	0	0	0
$T_{ext\,h}$	hospital모델의 외부사건 처리시간	1	2	3	1	1
$T_{int\,h}$	hospital모델의 내부사건 처리시간	1	2	3	1	1
$T_{ext\,v}$	village모델의 외부사건 처리 시간	1	2	3	1	1
$T_{int\,v}$	village모델의 내부사건 처리시간	1	2	3	1	1



〈그림 13〉 실험결과

실험 결과 그림 13에 나타난 바와 같이 일반적인 경우 노드의 수가 증가함에 따라 전체 시뮬레이션 시간은 감소되었다. 특히 노드의 수가 비교적 적은 부분(노드의 수가 2~4개 정도)에서는 시뮬레이션 시간이 크게 줄어들었다. 그리고 노드 수가 증가하면서 감소추세는 둔화되었는데 이는 노드의 수가 증가하면서 노드간의 통신을 위한 오버헤드가 더욱 더 커지기 때문이다. 그리고 계산 시간과 통신 시간의 비율도 속도 향상에 크게 영향을 미쳤다. 계산 시간의 비중이 커짐에 따라 더 큰 속도 향상이 있었다. 특히 노드간의 통신 시간이 아주 클 경우에는 노드의 수가 증가할 때 오히려 시뮬레이션 시간이 증가하는 경우도 있었다.

5. 결론

본 연구에서는 병렬 분산 환경에서는 DEVS추상화 시뮬레이터인 P-DEVSIM++의 시뮬레이션 동작을 모델링하고 그것을 성능 측정 시뮬레이터로 구현하였다. 그것을 위하여 P-DEVSIM++의 시뮬레이션 동작을 시뮬레이션 환경에 독립적인 부분과 의존적인 부분으로 나누었다. 환경에 독립적인 부분의 시뮬레이션 동작은 추상화 시뮬레이터의 동작에 의하여 결정되므로 추상화 시뮬레이터들이 여러 종류의 메시지를 받았을 때의 처리 과정을 모델링하였다. 환경에 의존적인 부분은 새로운 클래스 Psim을 도입하여 모델링하였다. 특히 두 노드간의 통신을 모델링하기 위하여 실제 메시지와 확인 메시지의 두 종류의 메시지를 도입하고 통신을 위한 Psim의 자료구조들도 정의하였다.

구현된 성능 측정 시뮬레이터를 이용하여 P-DEVSIM++의 병렬처리 성능을 측정하였다. 그것을 위한 벤치마크 모델로는 병렬 이산사건 시뮬레이션의 벤치마크로 널리 알려진 CHCS(Columbian Health Care System)모델을 이용하였다. 시뮬레이션에 사용된 노드의 수, 두 노드간의 통신 시간, 각 모델의 내부·외부사건 처리 시간을 바꾸어 가며 반복해서 실험하였다. 일반적인 경우에는 시뮬레이션에 사용된 노드의 수가 증가하면서 전체 시뮬레이션을 하는 데에 걸리는 시간이 감소하였다. 특히 노드의 수가 비교적 적을 경우에는 매우 큰 속도의 증가가 있었다. 그러나 노드 수가 많을 경우에는 통신 오버헤드가 커져서 큰 속도의 변화가 없었다. 그리고 통신 시간과 내부·외부사건 처리 시간의 값에 따라 노드 수에 다른 시뮬레이션 속도의 변화가 크게 변화하였다. 특히 통신시간이 매우 클 경우에는 노드의 수가 증가할 때 오히려 시뮬레이션 시간이 더 길게 나타난 경우도 있었다. 현재 P-DEVSIM++의 병렬처리 성능을 극대화하기 위한 대응 알고리즘에 대한 연구가 진행 중이다.

[참고문헌]

[1] 성영락, 정성훈, 김탁곤, 박규호, "병렬 분산 환경에서의 DEVS 형식론의 구현," 「한국시뮬레이션학회 논문지」, 1권, 1호, pp. 64-76, 1992.

[2] D. Baezner, G. Lomow, and B. W. Unger, "Sim++:

The transition to distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series, 1990.

[3] A. I. Concepcion, "Distributed simulation on multiprocessor:Specification, design and architecture," PhD dissertation, Wayne State University, 1985.

[4] A. I. Concepcion and B. P. Zeigler, "DEVS formalism: A framework for hierarchical model development," *IEEE Trans. on Software Engineering*, vol. 14, no. 2, pp. 228-341, Feb. 1988.

[5] R. Jain, *The Art of Computer Systems Performance Analysis*.

[6] T. G. Kim and S. B. Park, "The DEVS formalism: Hierarchical modular systems specification in C++," in *Proceedings in European Simulation Multiconference*, 1992.

[7] G. Lomow, J. Cleary, B. W. Unger, and D. West, "A performance of conservative algorithms," in *Proceedings of the SCS Multiconference on Distributed Simulation*, Simulation Series, 1988.

[8] S. B. Park, "DEVSIM++:A semantics based environment for object-oriented modeling of discrete event systems," thesis, KAIST, 1993.

[9] B. P. Zeigler, *Theory of Modelling and Simulation*. John Wiley, 1976.

[10] B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[11] B. P Zeigler and G. Zhang, "Mapping hierarchical discrete event models to multiprocessor systems: Concepts, algorithm, and simulation," *J. Parallel and Distributed Computing*, vol. 10, no. 3, pp. 271-281, July 1990.

[12] D.-K. Baik, "Performance evaluation of hierarchical simalators : Distributed model tiransformations and mappings," PhD dissertation, Wayne State Vniversity, 1985.

저자소개

성영락

1989년 2월 한양대학교 공과대학 전자공학과 졸업(공학사)
1991년 2월 한국과학기술원 전기 및 전자공학과 졸업(공학석사)
1991년 3월~현재 한국과학기술원 전기 및 전자공학과 박사과정
관심분야 : 시뮬레이션 형식론, 병렬 시뮬레이션, 병렬처리

김탁곤

1975년 2월 부산대학교 전자공학과 졸업(공학사)
1980년 2월 경북대학교 전자공학과 졸업(공학석사)
1988년 5월 아리조나대학교 전기 및 전산공학과 졸업(공학박사)
1975년 2월~1977년 6월 육군 통신장교
1980년 9월~1983년 1월 부산수산대학교 전자통신공학과 전임강사
1987년 8월 1989년 7월 아리조나 환경연구소 연구엔지니어
1989년 8월~1991년 8월 캔사스대학교 전기 및 전산공학과 조교수
1991년 9월~1993년 8월 한국과학기술원 전기 및 전자공학과 조교수
1993년 9월~현재 한국과학기술원 전기 및 전자공학과 부교수
본학회 논문지 편집위원장
관심분야: 모델링이론, 병렬/지능형 시뮬레이션 환경, 컴퓨터 시스템 등

박규호

1973년 서울대학교 전자공학과 졸업(공학사)
1975년 한국과학기술원 전기 및 전자공학과 졸업(공학석사)
1983년 프랑스 파리 11 대학 졸업(공학박사)
1975년~1978년 동양정밀 개발과장
1983년~1988년 한국과학기술원 전기 및 전자공학과 조교수
1988년~1922년 한국과학기술원 전기 및 전자공학과 부교수
1992년~현재 한국과학기술원 전기 및 전자공학과 교수
관심분야 : 병렬처리, 컴퓨터 구조, 컴퓨터 비젼