Journal of the Korean Institute of Industrial Engineers
Vol. 32, No. 3, pp. 191-199, September 2006.

비즈니스 프로세스 관리를 위한 분산 실행 모형 설계

허 원 창†

인하대학교 경영학부

Design of a Distributed Enactment Model for Business Process Management

Wonchang Hur

College of Business Administration, Inha University, Incheon 402-751

Effective management of business processes is a crucial issue to every enterprise in e-business environment. What's needed is a new framework of applications that can automatically manage distributed and heterogeneous business processes that span multiple functions of a company. In this paper, we propose technical design of a new enactment model that can coordinate such business process that involves multiple functional units or even multiple companies. In our approach, a process model is decomposed into several structural units, called 'process block', according to their procedural characteristics. Each of them is controlled by autonomous enactment units that can communicate with each other using a mutually agreed coordination protocol. The protocol takes the use of 'associative communication' concept, which allows the autonomy for each unit and secure the correctness of process execution.

***Keywords*: **Business Process Management, Distributed Enactment Model, Coordination

1. 서 론

1.1 연구의 중요성

BPM(Business Process Management)은 비즈니스 프로세스 관리를 통해 기업의 경쟁력을 확보하고자 하는 경영혁신 개념이다. 급속한 정보기술(information technology)의 발달은 이러한 개념을 지원하는 정보시스템인 BPM 시스템을 등장시켰다. BPM 시스템은 비즈니스 프로세스의 정의, 자동화 및 실행 통제, 진행 과정의 모니터링과 결과의 분석 및 평가에 이르는 과정을 효과적으로 지원한다.

근래 e-비즈니스로 인한 비즈니스 환경의 변화는 기업 간의 활발한 업무 교류를 촉발하였고, 이에 따라 BPM의 개념은 이제 기업 간의 협력적 비즈니스 프로세스 관리에 초점이 맞추어 지고 있다.

일반적으로 대부분의 BPM 시스템들은 기업 내부의 업무 자동화에 초점을 맞추고 있다(Ader *et al*., 2002). 이들은 기술적으로 별도의 실행서비스(enactment service)를 이용하는데, 그 관리 영역은 대개 조직의 경계를 넘지 못한다. 이러한 상황은, 조직 간 혹은 기업 간 협업을 요구하는 비즈니스 프로세스의 모형화와, 이의 실행을 통제 할 수 있는 새로운 BPM 시스템을 필요로 하고 있다.

1.2 배 경

BPM 시스템의 기능은 <Figure 1>에서와 같이 일반적으로 설계시점(build-time) 기능과 실행시점(run-time) 기능으로 나뉜다.

본 연구는 BPM의 각 기능영역에 대하여 각각 세 가지의 구성모형(component model)을 포함하는 기술 프레임워크를 기반으로 전개된다.

설계시점 기능영역에는 비즈니스 프로세스의 구조 및 논리

---

† 연락저자 : 허원창 교수, 402-751 인천광역시 남구 용현동 253번지 인하대학교 경영학부, Tel : 032-860-7733, Fax : 032-866-6877, E-mail : wchur@inha.ac.kr

2006년 1월 접수; 2006년 3월 수정본 접수; 2006년 4월 게재 확정.

적 수행 절차를 표현하는 프로세스 모형(Process Model)과 비즈니스 프로세스의 수행에 필요한 문서, 응용프로그램, 인적자원, 조직체계 등 조직에서 활용되는 자원들을 표현하는 자원 모형(Resource Model), 그리고 프로세스의 수행도 분석 및 평가 방식을 정의하는 분석 모형(Analysis Model)이 필요하다(Kim C. *et al*., 2001).

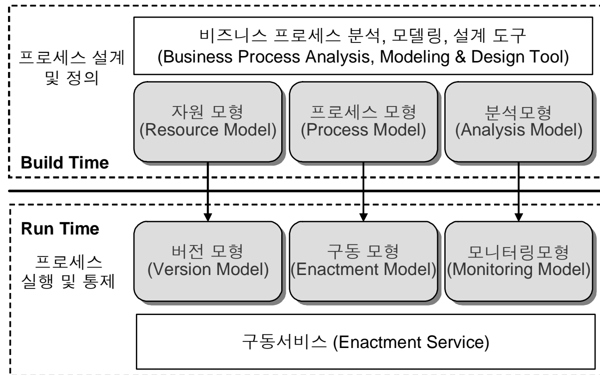

Figure 1. BPM framework.

설계시점의 각 구성모형에 대응하여, 실행시점 기능영역의 구성모형들이 필요하다. 이에는, 프로세스 모형에 따라 비즈니스 프로세스의 실행 방식을 기술하는 실행 모형(Enactment Model)과, 자원 모형에 표현된 자원들의 프로세스 실행에 따른 변경 이력의 관리 방식을 기술하는 버전 모형(Version Model)(Bae *et al*., 2001), 그리고 분석 모형에 따라 프로세스의 현재 상태, 실행 결과 및 자원들의 변경 내용 등을 실시간으로 모니터링 하는 방식을 기술하는 모니터링 모형(Monitoring Model)(Hur *et al*., 2003)이 있다.

1.3 연구의 목표

본 논문에서는 이러한 구성 모형들 중 기업 간 비즈니스 프로세스의 표현과 실행 방식을 기술하는 새로운 구조 모형과 실행 모형을 제시한다.

비즈니스 프로세스의 실행과 관련하여 다양한 연구가 수행되었다. Casati *et al*.(1996, 2000)는 비즈니스 프로세스의 패턴을 통해 활동규칙(active rule)을 유도하여, 프로세스를 실행하는 방식을 제안하였다. Dayal(1990)은 데이터베이스의 트리거(trigger)를 사용하여 비즈니스 트랜잭션과 같이 오랜 시간동안 지속되는 트랜잭션의 수행을 통제하려고 하였다. 하지만 이러한 시도들은 모두 중앙집중형 데이터베이스 시스템에 기반을 둔 실행모형으로, 분산된 실행서비스를 제공하지 못하였다.

이 밖에 모바일 에이전트를 이용한 방법이나(Cabri *et al*., 2000), 이벤트 기반의 구조(Cugola *et al*., 2001)를 사용하여 프로세스를 관리하려는 시도, XML과 웹서버를 이용한 시도(Tolksdorf, 2002)가 있었으나, 이들은 모두 분산된 실행서비스들 간의 긴밀한 결합을 요구하였다.

<Figure 2>는 기업 간 일반적인 주문처리 프로세스를 나타낸 것으로, 하나의 협업 프로세스는 주문요청, 주문처리, 송장발송, 송장확인 의 네 개의 단위 프로세스 모형으로 분해되어 물리적으로 분산되어 실행된다.

본 연구에서는 이러한 분산 실행 환경에서, 각 단위 프로세스 모듈간의 실행 의존관계를 통제할 수 있는 실행모형을 제안한다.

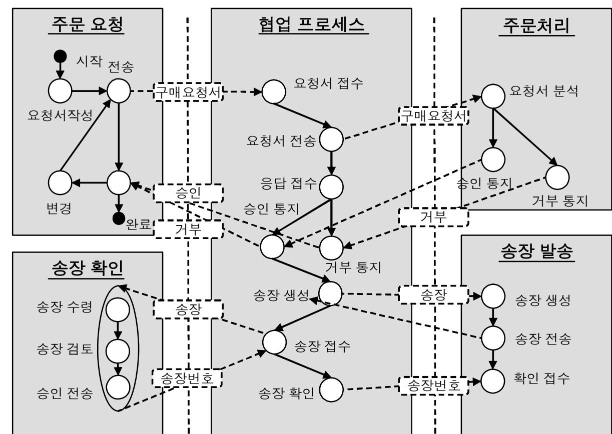

Figure 2. Order Fulfillment Process (Jung *et al*., 2004).

본 연구의 실행모형은 프로세스 모형을 구조단위(structural unit)로 분해하고, 각 구조단위는 공유된 데이터 공간을 통한 느슨한 결합(loosely coupled)으로 통신하여, 각 실행과정의 독립성을 보장하면서도 전체 프로세스가 올바르게 실행될 수 있도록 통제된다. 이는 프로세스 모형이 반드시 중앙의 실행서비스에 의해 독점적으로 관리 및 실행되어야 하는 제약을 극복하고, 프로세스 모형의 동적인 변경, 확장 및 외부 비즈니스 프로세스와의 실시간 연결 등이 빈번히 요구되는 협업 중심의 비즈니스 환경을 효과적으로 지원할 수 있다.

2. 프로세스 구조 모형

2.1 프로세스 블록 모형

본 연구에서는 '프로세스 블록(process block)'을 이용한 비즈니스 프로세스 표현 방식을 사용한다. 프로세스 블록은 다수의 업무들이 특정한 실행 패턴으로 연결되도록 구성한 업무들의 집합으로 비즈니스 프로세스 모델링을 위한 표현단위로 사용된다(Gokkoca *et al*., 1997; Bae *et al*., 2004).

하나의 프로세스 블록은 다른 프로세스 블록들을 이용하여 구성할 수 있기 때문에, 최종적으로는 <Figure 3>과 같이 '블록나무(block tree)' 형태로 비즈니스 프로세스를 정의할 수 있다. 특히, Bae *et al*.(2004)는 일반적으로 사용되는 네트워크 형태의 표현방식을 블록나무로 변환하는 알고리듬을 제안하여 본 모형의 유용성을 입증하였다.

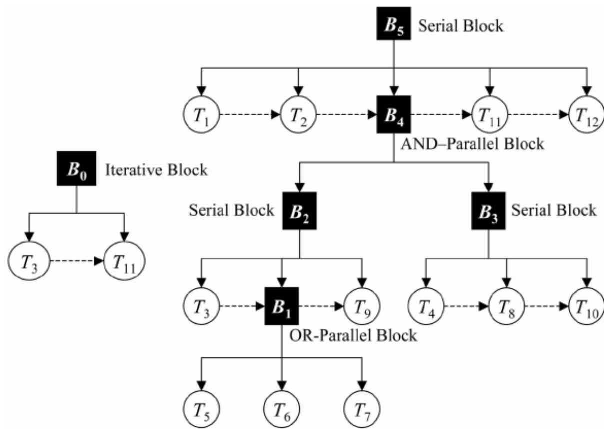

Figure 3. Process Block Tree (Bae *et al.*, 2003).

2.2 실행패턴과 상태전이 의존관계

프로세스 블록은 여러 개의 구성업무를 포함한다. 일반적으로 한 구성업무는 하나의 트랜잭션으로 표현되며, 트랜잭션은 <Figure 4>와 같이 시작(begin), 완료(commit), 실패(abort)의 세 가지 상태전이 이벤트의 발생에 따라 준비(ready), 실행(running), 실패(aborted), 정상완료(committed) 중 하나의 상태를 갖게 된다.

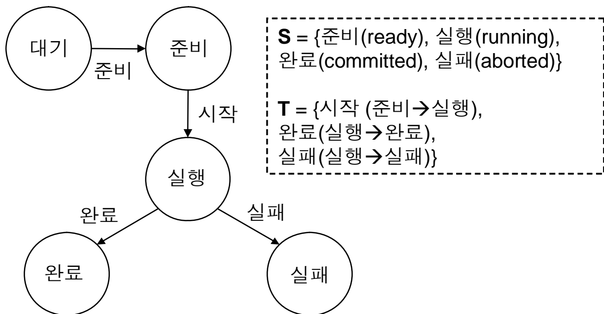

Figure 4. State Transition Diagram.

많은 수의 병렬적인 트랜잭션이 존재하는 복잡한 응용프로그램의 경우 트랜잭션의 상태전이 간에 다양한 제약조건이 존재하는데, 상태전이 의존관계는 이러한 제약조건을 표현하기 위해 개발된 확장 트랜잭션 모형에서 사용되는 개념이다. 대표적인 확장 트랜잭션 모형인 ACTA 형식론(formalism)에서, Chrysanthis *et al.*(1995)은 <Figure 5>에서 보는 바와 같이 9개의 상태전이 의존관계를 제시하였다.

그림에서 H는 이미 수행된 트랜잭션 이력(history)을 의미하며, '<' 기호는 시간적 선후관계를 의미한다. 예를 들어, CD(Commit Dependency)는 트랜잭션 t_i, t_j의 완료(commit) 이벤트간의 의존관계를 정의한 것으로, t_i와 t_j가 모두 완료되었을 때, 시간적으로 t_i가 t_j보다 앞서야 한다는 제약조건을 의미한다.

Gokkoca *et al.*(1997)에 의하면 ACTA 형식론에서 사용된 의존관계들은 다음의 정의와 같은 기본적인 형태의 상태전이 의존관계를 기반으로 하여 다시 표현할 수 있다.

[정의-상태전이 의존관계] 두 상태전이 이벤트 $e_1, e_2 \in$ {'begin', 'commit', 'abort'}에 대하여 다음의 두 조건을 만족하면, e_1, e_2는 트랜잭션 의존관계가 존재한다고 하며, $e_1 \rightarrow e_2$와 같이 표시한다.
1. e_1의 발생은 e_2의 발생을 의미(involve)함.
2. e_1의 발생은 e_2의 발생에 시간적으로 선행함.

정의에서 첫 번째 조건은, e_1이 발생하였을 경우, 과거, 혹은 미래의 어느 시점에 e_2가 발생해야 한다는 것을 의미하며, 두 번째 조건은 두 이벤트의 시간적 선후관계를 의미한다.

Dependency Relation	Name	Definition
t_j CD t_i	Commit Dependency	(Commit $t_j \in H$) $\Rightarrow$ ((Commit $t_i \in H$) $\Rightarrow$ (Commit t_i < Commit t_j))
t_j SCD t_i	Strong-Commit Dependency	(Commit $t_i \in H$) $\Rightarrow$ (Commit $t_j \in H$)
t_j AD t_i	Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ (Abort $t_j \in H$)
t_j WAD t_i	Weak-Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ ((Commit t_j < Abort t_i) $\Rightarrow$ (Abort $t_j \in H$))
t_j BD t_i	Begin Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Begin t_i < Begin t_j)
t_j BCD t_i	Begin-on-Commit Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Commit t_i < Begin t_j)
t_j BAD t_i	Begin-on-Abort Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Abort t_i < Begin t_j)
t_j CAD t_i	Commit-on-Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ (Commit $t_j \in H$)
t_j WCD t_i	Week-Begin-on-Commit Dependency	(Begin $t_j \in H$) $\Rightarrow$ ((Commit $t_i \in H$) $\Rightarrow$ (Commit t_i< Begin t_j))

Figure 5. Dependency Relations in ACTA formalism (Chrysanthis *et al.*, 1995; Bae *et al.*, 2004).

하나의 프로세스 블록은 특정한 실행패턴을 갖는다. 실행패턴이란 해당 프로세스 블록의 구성업무들이 실행되는 절차를 규정한 것이다. 프로세스 블록의 실행패턴은 구성업무 트랜잭션간의 상태전이 의존관계를 규정함으로써 정의할 수 있다. 예를 들어, <Figure 5>는 n개의 구성업무가 병렬적으로 실행되는 OR 병렬블록의 실행과정에서 상태전이 의존관계를 표현한 것이다. Bae *et al.*(2004)은 직렬, 순차, AND-병렬, OR-병렬 등 총 6 종류의 실행패턴을 갖는 프로세스 블록을 트랜잭션 의존관계를 이용하여 정의하였다.

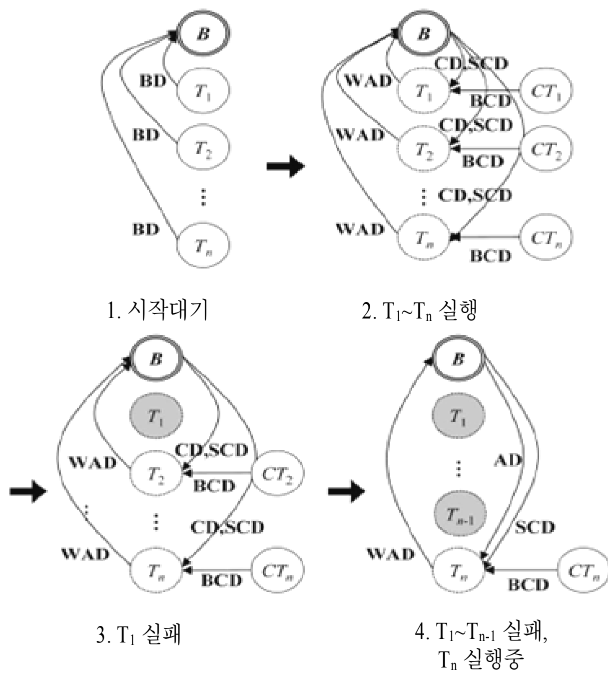

Figure 6. Dependency Relations in OR-parallel block (Bae *et al*., 2004).

3. 프로세스 실행 모형

프로세스 블록을 이용하여 표현된 비즈니스 프로세스의 실행에 있어서 가장 중요한 문제는<Figure 6>과 같은 상태전이 의존관계를 어떻게 통제할 것인가의 문제이다. 본 연구에서는 분산 및 병렬 처리를 위하여 도입된 프로그래밍 모델인 코디네이션 모형(Gelernter *et al*., 1986)을 이용하여 프로세스 블록의 실행을 통제하는 실행모형을 설계하였다.

3.1 코디네이션 모형

코디네이션 모형에서는 분산된 실행 단위들이 공유된 데이터공간을 통해 특정한 형식의 데이터 객체를, 미리 정의된 기본적인 연산을 통하여 생성하고 소비하면서 정보를 교환하고 문제를 해결한다. 각 요소에 대하여 자세히 살펴보도록 한다.

3.1.1 투플과 투플공간(tuple space)

투플공간은 '투플(tuple)'이라고 하는 단위 데이터들이 생성되는 데이터 공간이다. 하나의 투플은 특정한 데이터 타입을 갖는 여러 개의 필드로 구성된다. 특히, 각 필드는 '?' 기호로 표현되는 변수나, 특정한 연산을 수행하는 함수를 사용하여 표현할 수도 있다.

변수를 필드로 포함하는 투플을 형식(formal)투플 이라고 하여 일반(actual) 투플과 구분한다. 함수를 필드로 포함하는 투플을 특별히 능동형 투플이라고 하는데, 이 경우 투플공간에는 함수를 수행한 결과, 즉 함수가 반환한 값을 이용하여 투플이 생성된다. 예를 들어, 투플 [π,sin(π),cos(π)]는 두 개의 삼각함수를 실행시키며, 그 실행이 완료된 시점에는 [π, 0, -1]로 변환되어 저장된다.

Table 1. Tuple format and examples

투플의 형식
[T, e, t] T : 트랜잭션, e : 이벤트, t : 시간 ex) [T_1, begin, 3], [T_2, commit, ?t]

<Table 1>은 프로세스 실행모형에서 사용되는 투플의 형식이다. 앞서 기술한 바와 같이 표에서 트랜잭션은 해당 프로세스 블록을 구성하는 구성업무를 의미하며, 이는 프로세스 블록 자체가 될 수도 있다. 예를 들어, [B_1, begin, t]은 프로세스 블록 B_1이 시점 t에 'begin' 이벤트를 발생하였다는 것을 의미한다.

3.1.2 투플연산(tuple operation)

각 트랜잭션들은 투플공간에 투플을 읽고 쓰기 위하여, <Table 2>에 보는 바와 같이 'publish', 'subscribe', 'read'의 세 가지 연산을 사용한다.

Table 2. Tuple operations

연산	예	비고
publish	publish([T,'begin',3])	투플생성
subscribe	subscribe([?X,'begin',3])	변수할당, 투플삭제
read	read([?X,'begin',3])	변수할당, 투플비삭제

'publish'는 투플공간에 새로운 투플을 생성하는데 사용된다. 'subscribe'와 'read'는 변수가 필드로 포함된 투플을 사용하여 변수 값을 읽어오는데 사용된다. 표의 예에서 'subscribe'는 투플공간에서 시점 3에 'begin' 이벤트를 발생한 트랜잭션을 찾아 변수 X에 할당한다. 여기서 주목할 점은 두 연산 모두 변수에 값이 할당될 때까지 완료되지 않고 대기한다는 점이다. 즉 표의 예에서 두 연산이 종료되기 위해서는 시점 3에 'begin' 이벤트를 발생시킨 트랜잭션이 존재해야 한다. 'read'는 'subscribe'와 달리 변수에 값을 할당한 후 해당 투플을 삭제하지 않는다.

구체적으로는, 'subscribe'와 'read'연산을 수행하기 위해서는 변수를 포함한 투플과 매칭(matching)되는 일반 투플을 투플공간에서 검색한다. 투플간의 매칭은 다음 정의의 규칙을 따른다. 정의에 따르면, <Table 2>의 예에서 투플 [T, 'begin', 3]은 [?X, 'begin', 3]과 매칭규칙을 만족한다.

[정의-매칭규칙] 일반투플 a = [a_1, a_2, …, a_n]와 형식투플 f =

$[f_1, f_2, \ldots, f_m]$ (f_1~f_m 중 적어도 하나는 변수)가 다음의 두 조건을 만족할 경우 투플 a와 f는 매칭 된다고 한다.
1. n = m,
2. for all i, f_i가 변수가 아니면, $a_i = f_i$이다.

이와 같이 프로세스 블록을 구성하는 트랜잭션들은 위의 세 가지 연산과 매칭규칙을 사용하여 투플공간을 통해 특정한 데이터를 교환할 수 있는데, 이러한 방식의 데이터 교환방식을 연관적(associative) 혹은 생성적(generative) 통신이라 한다.

연관적 통신의 주목할 만한 특징은 트랜잭션간의 데이터 교환을 위해 이들이 시/공간적으로 결합될 필요가 없다는 점이다. 이는 트랜잭션들이 서로를 직접적으로 호출하기 위해 물리적 주소를 알아야 하거나, 실행 상태를 동기화시켜야 하는 과정이 불필요하다는 것을 의미하는 것으로써, 각 트랜잭션의 독립성을 최대한 보장하면서도 전체적으로 원활한 정보공유를 가능하게 하는 장점이 있다.

3.1.3 상태전이 순서화(ordering)

본 연구에서는 각 프로세스 블록의 구성업무들이 투플연산을 통하여 트랜잭션의 연산을 수행함으로써, 업무간의 의존관계를 통제하는 방법을 제시한다. 다음의 순서화 프로토콜은 투플연산을 통해 트랜잭션의 상태전이가 순서화하는 방식을 설명한다.

[순서화 프로토콜] 실행 중인 트랜잭션 T_1, T_2에 대해서, T_2가 최초로 이벤트 e_2를 발생하려고 할 때, 다음 순서로 투플연산을 수행한다.
P1. read($[T_1, e_1, ?t]$);
P2. 상태전이 e_2를 수행한다.
P3. publish ($[T_2, e_2$, ‘now’]);

순서화 프로토콜은 $T_1.e_1 \rightarrow T_2.e_2$의 의존관계를 보장한다. 이의 증명은 투플연산 ‘read’의 속성에 의해서 명백하다. 트랜잭션 T_2는 상태전이 이벤트 e_2를 발생하기 전에, 트랜잭션 T_1의 상태전이 이벤트 e_1에 대하여 ‘read’ 연산을 수행한다. 따라서 투플공간에 트랜잭션 T_1의 상태전이 이벤트 e_1이 존재할 때 까지, e_2를 ‘publish’ 하지 않는다. 즉, T_2가 e_2를 ‘publish’ 했다는 것은, 이미 T_1이 e_1을 ‘publish’ 했다는 것을 의미한다.

<Figure 7>은 프로세스 블록 T_1의 ‘완료(commit)’와 T_2의 ‘완료’ 이벤트 사이에 의존관계가 존재한다고 할 때, T_1, T_2가 순서화 프로토콜에 따라 이를 통제하는 과정을 나타낸다.

그림에서, T_1은 시점 tx2에 ‘publish([T_1, ‘commit’, tx2])’ 연산을 수행하고, T2는 시점 ty1에 ‘read([T_1,commit’,?t]’ 연산을 수행한다. 의존관계를 만족시키기 위해서는 tx1 > ty2를 만족해야 한다. 그런데, ‘read’ 연산은 매칭규칙을 만족하는 투플 [T_1, ‘commit’, tx2]가 투플공간에 생성되는 시점인 tx2에 완료되므로, t = tx2가 되며, tx1 > tx2 = t > ty1 > ty2로 인해 의존관계를 만족한다.

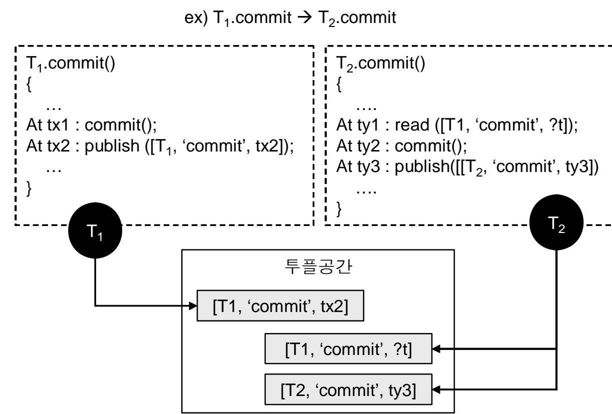

Figure 7. Controlling dependency relations using tuple operations.

3.1.4 상태전이 동기화(synchronization)

상태전이 동기화는 다수의 트랜잭션의 상태전이를 시간적으로 동일한 시점에 발생하도록 통제하는 것인데, 이는 운영체제(operating system)에서 사용되는 세마포(semaphore)의 개념과 유사하다. 세마포는 투플공간과 투플연산을 통하여 쉽게 구현될 수 있다. 다음은 투플연산을 이용한 동기화 프로토콜을 설명한다.

[동기화 프로토콜] 실행 중인 트랜잭션 T_1, T_2에 대해서, T_2가 최초로 상태전이 e_2를 수행하려고 할 때, 다음 순서로 투플연산을 수행한다.
P1. publish($[T_2, e_2$, now]) ;
P2. subscribe($[T_1, e_1, ?t]$) ;
P3. 상태전이 e2를 수행한다.

동기화 프로토콜은 $T_1.e_1 \leftrightarrow T_2.e_2$를 만족한다. 이의 증명 역시 순서화 프로토콜에서와 마찬가지로 ‘subscribe’ 연산의 속성에 의해 명백하다. 트랜잭션 T_2는 P1에서 자신이 수행하고자 하는 상태전이를 ‘publish’한다. 이는 자신이 동기화될 준비가 되어있다는 것을 T_1에게 알리는 역할을 한다. P2에서는 ‘subscribe’연산을 통해 T_1이 상태전이 준비가 되어있는 지를 확인한다. 이 두 과정을 통해 P3에서 행해지는 두 트랜잭션의 상태 전이는 동일한 시점이 시작될 수 있다

3.2 프로세스 블록의 실행 절차

본 절에서는 프로세스 블록을 구성하는 업무 트랜잭션의 작동방식을 제시하고, 이를 통해 해당 프로세스 블록이 정의된 실행 패턴에 따라 실행되는 절차를 설명한다.

3.2.1 의존관계의 표현

앞 절에서 기술한 바와 같이, 프로세스 블록은 특정한 실행 패턴을 가지며, 그 실행 패턴은 프로세스 블록을 구성하는 업

무 트랜잭션 간의 상태전의 의존관계로 표현할 수 있었다.

다음의 <Table 3>은 프로세스 모델링에 일반적으로 사용되는 직렬, AND병렬, OR병렬, XOR병렬의 네 가지 실행패턴의 프로세스 블록들에 대하여, 트랜잭션간의 상태전의 의존관계를 표현한 것이다. 예를 들어, AND 병렬블록의 ‘완료’는 모든 구성 트랜잭션의 ‘완료’ 이벤트와 의존관계를 가지고 있다. 따라서 모든 구성 트랜잭션이 완료해야만AND 병렬블록이 완료할 수 있다.

다양한 형태의 의존관계를 사용하여 표에 제시된 기본적인 실행패턴으로부터 보다 확장된 형태의 프로세스 모형을 설계하는 것이 가능하다.

Table 3. Dependency Relations for process blocks

실행패턴	의존관계		
직렬 블록	B	begin	없음
		commit	T_n.commit
		abort	$\exists_i(T_i.abort)$
	T_i	begin	B.begin ∧ T_{i-1}.begin (i>1)
		commit	T_{i-1}.commit (i>1)
		abort	없음
AND 병렬 블록	B	begin	없음
		commit	$\forall_i(T_i.commit)$
		abort	$\exists_i(T_i.abort)$
	T_i	begin	B.begin
		commit	없음
		abort	없음
OR 병렬 블록	B	begin	없음
		commit	$\exists_i(T_i.commit)$
		abort	$\forall_i(T_i.abort)$
	T_i	begin	B.begin
		commit	없음
		abort	없음
XOR 병렬 블록	B	begin	없음
		commit	$\exists_i(T_i.commit)$
		abort	$\forall_i(T_i.abort)$
	T_i	begin	B.begin ∧ T_{i-1}.abort (i>1)
		commit	없음
		abort	없음

3.2.2 프로세스 블록의 실행 규약

프로세스 블록은 각자의 실행패턴에 의해 규정된 상태전이 의존관계를 유지하면서, 프로세스 블록과 그 구성 트랜잭션들이 실행될 수 있도록 하기 위해 특정한 실행 규약을 따른다

이러한 실행 규약은 앞서 기술한 투플연산을 통한 순서화 프로토콜과 동기화 프로토콜에 기반을 두어 설계되었다. <Table 4>는 프로세스 블록을 포함한 프로세스 모형을 구성하는 모든 업무 트랜잭션들이 실행되는 절차를 기술한 것이다.

Table 4. Execution procedure

1. ‘시작’ 의존관계 확인
2. T_i을 시작하고, ‘시작’ 이벤트 ‘publish’
3. 트랜잭션의 주요부분 수행

3의 실행이 성공했을 경우

4. ‘완료’ 의존관계 확인
5. T_i를 완료하고 ‘완료’ 이벤트 ‘publish’

그렇지 않은 경우

6. ‘실패’ 의존관계 확인
7. T_i를 종료하고 ‘실패’ 이벤트 ‘publish’

트랜잭션은 실행과 동시에 우선 ‘시작’ 의존관계를 확인한다. 의존관계가 만족되면 ‘시작’ 상태전이를 투플공간에 ‘publish’ 연산을 통해 기록하고 ‘실행‘ 상태가 된다. ‘실행‘ 상태에서는 해당 트랜잭션의 고유한 업무를 수행한 후, 그 결과에 따라 ’완료‘와 ’실패‘의 여부가 결정된다. ‘완료‘의 경우 ‘완료’ 의존관계가 만족되면 ‘완료’ 상태전이를 진행하고 ‘실패’의 경우 마찬가지로 ‘실패’ 의존관계를 확인한 후 ‘실패’ 상태전이를 진행한다. 이와 같이 모든 상태전이는 앞서 기술한 투플연산을 통한 순서화 프로토콜을 따른다.

Table 5. Implementation of execution procedure

B	begin();
	publish[B, begin ,now];
	if(fork() == ‘primary’) { ....(1)
	while(PS ≠ {T1, ..., Tn}); { ....(2)
	subscribe[?X,commit,?t];
	if(X∈{T1, ..., Tn}) PS=PS∪X;
	}
	commit();
	publish[B,commit,now];
	}
	else {
	while(X∉{T1,...,Tn}){subscribe[?X,abort,?t];}..(3)
	abort();
	publish[B,abort,now];
	}
	terminate();

T_i	subscribe[B, begin, ?t]; ....(4)
	begin();
	publish[Ti, begin, now];
	result = execute_critical_section(); ....(5)
	if (result == ‘success’){
	commit();
	publish(Ti, commit, now);
	}
	else {
	abort();
	publish(Ti, abort, now);
	}

<Table 5>는 제시된 실행절차에 따라 AND 병렬블록과 그 구성업무 트랜잭션의 실행 절차를 구현한 의사코드(pseudo code)이다. 프로세스 블록의 경우 실행이 시작되면, (1)의 ‘fork’

명령을 통해 복제된 두 개의 실행 쓰레드(thread)를 생성하고, 각각은 구성 업무트랜잭션과의 의존관계에 따라 '완료'와 '실패'의 상황을 각각 대기한다. 즉 (2)의 'while' 구문에서는 'subscribe' 투플연산을 통해 AND 병렬블록의 완료조건인, 모든 구성업무 트랜잭션의 완료했는가를 검사한다. (3)의 'while' 구문에서는 역시 'subscribe' 투플연산을 통해 AND 병렬블록의 실패조건인, 실패한 구성업무가 있는가를 검사한다. 두 쓰레드 중 먼저 'while' 구문을 벗어나는, 즉 먼저 만족되는 의존관계에 의해 프로세스 블록의 '완료'와 '실패'가 결정된다.

한편 AND 병렬블록을 구성하는 구성업무 트랜잭션의 경우 (4)의 'subscribe' 투플연산을 통해 프로세스 블록의 실행이 시작을 대기한다. 트랜잭션이 시작되면, (5)에서 해당 트랜잭션의 주요한 작업을 수행한 후, 그 결과에 따라 '완료' 혹은 '실패' 상태전이를 진행한다.

<Figure 8>은 <Table 5>의 실행절차에 따라 의존관계가 보장되는 개념을 도시한다. 그림에서 AND 병렬블록 'B'는 상태전이 '완료'의 수행을 위해 <Table 3>의 의존관계에 따라 구성업무 트랜잭션의 '완료'를 대기한다. 전술한 바와 같이 'subscribe' 연산은 투플공간에 구성업무 트랜잭션의 완료를 의미하는 투플의 집합(PS[X])이 생성되는 것을 확인될 때까지 트랜잭션의 실행을 정지시킨다. 따라서 각 구성업무 트랜잭션이 실행 프로토콜에 따라 각자의 완료 상태를 'publish' 연산을 통해 투플공간에 등록할 때 까지 프로세스 블록의 실행은 중지된다. 이에 따라 병렬블록 B의 '완료' 상태전의 의존관계가 만족된다.

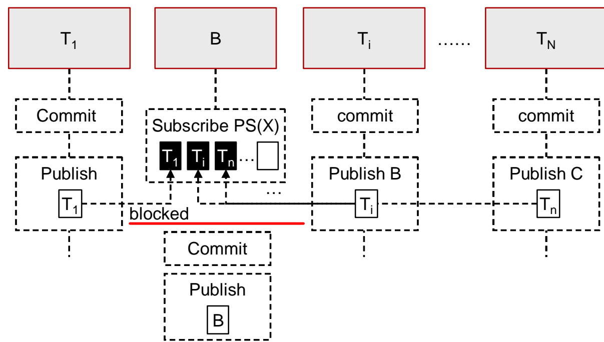

Figure 8. Controlling dependency relations in AND-parallel block.

3.2.3 프로세스 모형의 실행

프로세스 모형의 실행은 프로세스 블록나무의 최상위 프로세스 블록을 실행시키는 것으로 시작된다. 프로세스 블록이 실행되면, 실행패턴에 관계없이 구성업무 트랜잭션들을 실행시킨다. 구성업무 트랜잭션이 프로세스 블록일 경우, 그 블록에 포함된 구성업무 트랜잭션을 실행시킨다. 이러한 재귀적 방식으로 프로세스 모형에 포함된 모든 구성업무 트랜잭션이 실행된다.

구성업무 트랜잭션이 프로세스 블록인 경우에는, 우선 <Table 3>에서 트랜잭션에 해당하는 의존관계를 만족하도록 작동한다. 즉, 트랜잭션 의존관계가 프로세스 블록의 의존관계에 우선한다. 예를 들어, <Figure 3>의 B_1은 구성업무 트랜잭션 T_5, T_6, T_7을 포함하는 OR 병렬블록임과 동시에, 직렬블록 B_2에 포함되는 구성업무 트랜잭션이다. 프로세스 모형의 실행은 상위 프로세스 블록부터 시작되므로, B_1은 우선 B_2의 구성업무 트랜잭션으로서 의존관계를 만족하도록 작동하며, 일단 실행이 된 후에는 OR 병렬블록으로 작동한다. 프로세스 블록으로서 실행이 완료된 후에는 다시 구성업무 트랜잭션으로서 작동한다.

이러한 프로세스 모형의 실행 방식이 가지는 가장 큰 특징은, 프로세스 모형을 구성하는 모든 트랜잭션이 동시에 병렬적으로 진행된다는 것이다. 이는 기존의 실행모형들이 프로세스 모형의 실행을 전담하는 실행 서비스(enactment service)에 기반을 둔 중앙집중형 구조를 갖는 것과 큰 차이점을 지닌다. 즉, 각 프로세스 블록과 구성업무 트랜잭션은 의존관계를 통제하기 위해 서로를 호출하거나 명시적으로 연결될 필요가 없는 장점이 있다.

또한 이와 같은 실행 방식은 실행하려는 프로세스 모형의 구조에 독립적이다. 프로세스 모형이 아무리 복잡하더라도 그 실행 방식은 실행 프로토콜에 의해 일관적이라는 것을 의미한다.

4. 시스템 구조

<Figure 9>는 본 연구에서 제안하는 프로세스 모형과 그 실행 모형을 구현한 시스템의 구조를 도시한다.

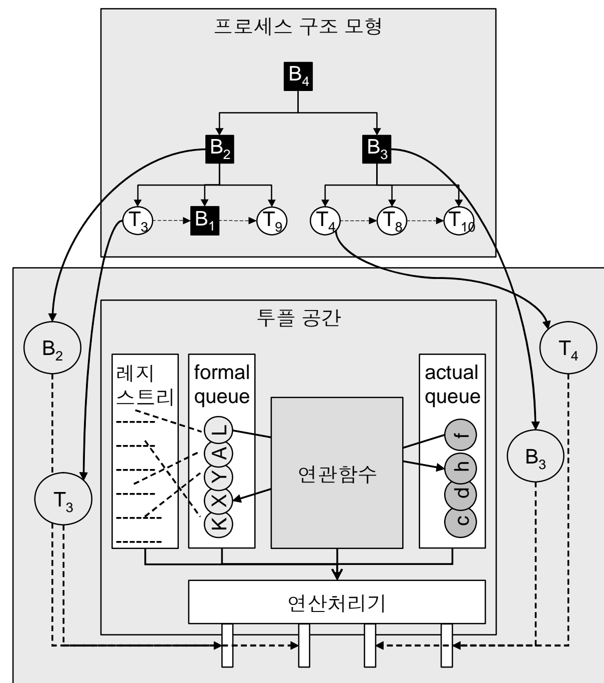

Figure 9. System Architecture.

4.1 투플공간의 구조

투플공간은 연관적 통신 방식을 지원하는 미들웨어(middleware)로 구현될 수 있다. 현재 Javaspace™나 IBM의 T-Space 등 쉽게 활용할 수 있는 개발용 라이브러리들이 나와 있다(Wyckoff *et al*., 1998).

<Figure 9>는 투플공간의 구현구조를 도식화한 것이다. 그림에서 보는 바와 같이 투플공간은 등록된 투플들을 저장하는 두 개의 대기행렬(formal queue, actual queue)과 트랜잭션들의 호출정보를 담는 레지스트리(registry), 투플간의 매칭 여부를 판단하는 연관함수(association function), 그리고 연산의 호출 및 관련 조정기와의 통신을 담당하는 연산처리기(operation handler)로 구성되어있다.

대기행렬은 투플공간을 통해 처리되는 투플들을 종류와 우선순위에 따라 구분하여 저장하고 관리하는 역할을 한다. 특히 변수를 포함한 투플을 저장하는 대기행렬(formal queue)은 각 투플을 'publish'한 트랜잭션들의 호출에 필요한 정보들을 레지스트리에 등록하여 관리한다.

연관함수는 새로운 투플에 대하여 대기행렬에 저장된 투플들과의 연관관계를 판단한다. 'publish' 연산으로 투플이 등록되는 경우 연관함수는 형식투플의 대기행렬을 검색하고, 'subscribe' 연산의 경우는 일반투플의 대기행렬을 검색한다.

마지막으로 연산처리기는 투플공간 외부의 트랜잭션들로부터 연산 호출을 접수하고 처리하는 역할을 한다. 호출된 연산의 종류에 따라 연관함수를 호출하고 그 결과에 따라 해당 투플을 적절한 대기행렬에 삽입하거나 혹은 연관화로 추출된 투플을 관련된 조정기로 전송하는 역할을 한다.

4.2 작동방식

투플공간은 연산처리기를 통해 여러 개의 네트워크 포트를 열고 외부 트랜잭션들로부터 연산 호출을 '감시(listening)'하는 상태로부터 시작된다. 특정 포트가 연산을 호출한 트랜잭션과 바인딩(binding) 되면 연산으로 생성된 투플의 처리 방식을 결정하기 위한 연관화 단계에 돌입하게 된다. 이 단계에서는 내장된 연관함수를 호출하며 그 결과에 따라 투플의 처리 방식이 결정된다.

연관화가 발생하지 않을 경우에는 투플을 등록하는 등록(enqueue) 단계로 진행한다. 이 단계에서는 등록할 투플의 종류 및 우선순위에 따라 해당 대기행렬에 투플을 삽입하고 데이터 구조를 갱신하게 된다. 특히 형식투플이 삽입될 경우에는 매칭된 투플을 등록한 트랜잭션의 호출에 필요한 정보가 레지스트리에 등록되며, 그 등록정보는 삽입된 투플과 연결되어 저장된다. 레지스트리에 자신의 정보를 등록한 트랜잭션은 활동을 멈추고 수면상태(sleep)가 된다. 등록단계가 끝나게 되면 조정공간은 다시 연산 처리기에 의해 감시단계로 돌아간다.

연관화가 발생되었을 경우에는 추출단계로 돌입한다. 이 단계에서는 매칭된 투플을 대기행렬에서 추출하고 해당 투플과 연관된 조정기의 호출 정보를 레지스트리로부터 추출하여 읽혀진 투플과 함께 연산 처리기에게 전달함으로써 라우팅(routing)단계에 이른다. 이 단계에서는 레지스트리에서 검색된 트랜잭션을 재실행(wake-up)시키고 연관된 투플을 전송한 후 다시 감시단계로 복귀한다.

5. 결 론

본 논문에서는 e-비즈니스 환경에서 조직 간, 혹은 기업 간 협력적 비즈니스 프로세스의 실행을 통제할 수 있는 새로운 프로세스 실행모형을 제안하였다. 이는 다음과 같은 특징과 장점을 가진다.

첫째, 실행서비스의 분산과 통제 – 프로세스 모형을 의미 단위인 프로세스 블록으로 분할하고, 그 실행을 독립적인 실행단위로 분할한다. 분할된 실행단위 간에는 직접적인 통신이 요구되지 않는다.

둘째, 프로세스 구조에 독립적인 실행 모형 – 각 프로세스 블록은 프로세스 구조에 독립적인 실행규칙에 따라 통제된다. 이러한 실행 방식은 프로세스 구조의 동적인 변경의 개념인 실시간 캡슐화(Kim *et al.*, 2000)를 가능하게 한다.

셋째, 구현 플랫폼에 독립적인 실행 모형– 실행 모형은 실행단위의 할당 방식, 수행 역할, 실행 규칙의 종류, 통신 방식 등에 따라 다양하게 변경 가능하다. 또한 표준화된 연산들을 바탕으로 하고 있어서, 동일한 비즈니스 프로세스의 실행 방식을 대상 플랫폼에 맞게 쉽게 변경할 수 있다.

이러한 특징은 협력적 프로세스의 실행과 모니터링을 효과적으로 지원할 수 있는 BPM 시스템의 개발을 가능하게 할 것이다.

참고문헌

Ader, M. (2002), Workflow Comparative Study, W&GS, Http:// www.wngs.com.

Bae, J., Bae, H., Kang, S., and Kim, Y. (2003), Automatic Control of Workflow Processes Using ECA Rules, *IEEE Transactions on Knowledge and Data Engineering*, **16**(8), 1010-1023.

Bae, H. and Kim, Y. (2001), A Document-Process Association Model for Workflow Management, *Computers in Industry*, **47**(2), 139-154.

Cabri, G. et al. (2000), MARS : A Programmable Coordination Architecture for Mobile Agents, *IEEE Internet Computing*, **4**(4), 26-35.

Casati, F. et al. (1996), Deriving Active Rules for Workflow Enactment, *Proc. 7th Int'l Conf. Database and Expert Systems Applications (DEXA '96)*, Zurich, Switzerland, 94-110,

Casati, F. et al. (2000), Using Patterns to Design Rules in Workflows, *IEEE Transaction on Software Engineering*, **26**(8), 760-785.

Chrysanthis, P. K. and Ramamritham, K. (1992), ACTA : The SAGA Continues, *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, edited by A. K. Elmagarmid,

Chapter 10.

Cugola, G., Nitto, E. D., and Fuggetta, A. (2001), The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS, *IEEE Transactions on Software Engineering*, **27**(9), 827-850.

Gelernter, D. (1985), Generative Communication in LINDA, *ACM Transaction on Programming Languages and Systems*, **7**(1), 80-112.

Dayal, U., Hsu, M., and Ladin, R. (1990), Organizing Long-running Activities with Triggers and Transactions, *Proceeding of the ACM SIGMOD*, 204-214.

Gokkoca, E. et al. (1997), Design and Implementation of a Distributed Workflow Enactment Service, *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, IEEE Computer Society, 89-98.

Jung, J. et al. (2004), Business Process Choreography for B2B Collaboration, *IEEE Internet Computing*, **8**(1), 37-45.

Kim, C. O., Jun, J., and Kim, S. S. (2001), Object-Oriented Business Process Modeling Contract-Collaboration Net Model, *Journal of the Korean Institute of Industrial Engineers*, **27**(1), 37-46.

Kim, Y. et al. (2000), WW-Flow : Web-Based Workflow Management with Runtime Encapsulation, *IEEE Internet Computing*, **4**(3), 55-64.

Tolksdorf, R. (2002), Workspaces : A Web-Based Workflow Management System, *IEEE Internet Computing*, **6**(5), 18-26.

Wyckoff, P. et al. (1998), T Spaces, *IBM Systems Journal*, **37**(3), 454-474.

Hur, W. et al. (2003), Customizable Workflow Monitoring, *Concurrent Engineering : Research and Application,* 313-325.