객체 지향모델인 Actor에서 최대 병렬성 지원의 순차 합성법

곽호영*, 김장형*, 안기중*

The Sequential Composition Method for Supporting Maximal Parallelism in Actor as Object-Oriented Model

*Kwak Ho-young**, *Kim Jang-hyung**, *Ahn Khi-jung**

Summary

In Actor model based on message-passing mechanism and parallel architecture, all commands in an Actor are processed concurrently. However, when sequential processing of commands must be preserved, it can be solved by creating actors called customers which are to do delegation, because concurrent compositions inherent in Actor, are not enough to do the sequential processing of commands.

In this paper, the dependency between expressions in the sequential composition of actor commands is analyzed, and the sequential composition method for supporting minimal sequentiality and maximal parallelism in the Actor model is presented.

서 론

미래의 컴퓨터 개발과 실세계의 빠른 변화에 적응력이 있는 개방 시스템의 개발에 있어서 병렬 처리 구조를 기본으로 하는 시스템 모델들에 관한 연구가 상당히 활발하게 진행되고 있다. 또한, 병렬 처리 구조에 적합한 병행(concurrent) 프로그래밍 언어와 분산 처리 언어 등의 프로그래밍 언어 분야도 동시에 연구되고 있으며, 이 프로그래밍 언어 분야 중에서도 객체 지향 언어에 관한 연구가 매우 흥미를 끌고 있다.

이와 같은 분야의 한 연구로 병렬 처리 구조와 병행 연산을 기본으로 하는 시스템 모델인 Actor가 발표되었는데, 이 Actor 모델은 파이프라인(pipeline) 방식에 의한 미세화(fine-grained) 병행성을 제공하고 있으며, 특수한 구현이나 응용에 국한되지 않은 일반적인 병렬 처리 이론 모델로 제시되었다.

그러나, 이 Actor 모델은 기본 개념이 병행 연산이지만 연산의 의존성이 부여된 순차 합성(sequential composition)의 경우에는 그 병렬성

* 공과대학 정보공학과

의 정도가 낮아지게 된다. 이러한 관점은 일반적인 병행 프로그래밍(concurrent programming) 언어에서도 동일하게 적용된다.

따라서, 본 논문에서는 Actor 모델에서 순차 합성을 처리하는 경우 최소 순차성(minimal sequentiality) 또는 최대 병렬성(maximal parallelism)의 보장을 위한 방법을 제시하였으며, 또한 〔정리 1〕과 〔정리 2〕〔Maekawa 87〕를 Actor 모델에 직접 적용한 알고리즘과 본 논문에서 제시한 알고리즘을 비교하였다. 본 논문에서는 알고리즘 자체에 중점을 둔 것이라기보다 병렬성의 향상을 위해 순차 합성시 발생되는 전달(forwarding) 기능만을 갖는 actor를 제거하여 프로세스의 수를 줄이고 그 처리의 단계수를 최소화하는 것에 목표를 두었다.

Actor 모델

1. 소 개

Actor 모델은 일반성(generality), 재구성성(reconfigurability) 및 확장성(extensibility)을 증시하는 객체 지향 프로그래밍의 미세화 병행성을 지원하는 모델이다. 그러나, 이 Actor 모델은 객체 지향 언어의 일반적 개념인 클래스-상속 기법을 사용하지 않고 delegation에 의해 객체의 처리 방법들을 각각의 객체(actor) 스스로 처리가 가능하도록 하고 있는 것이 특징이다. 이 모델에서 동작하는 Actor 언어로는 저수준 Actor 언어와 사용자가 작성하는 고수준 Actor 언어로 구분되는데, 고수준 Actor 언어는 Lisp와 유사한 표현을 구사하며 배정문을 사용하지 않는다.

또한 객체(object)의 개념을 지원하기 위하여 메시지 전달 기법에 의한 delegation 개념을 사용하고 있는데, 이 delegation은 프로세서 간에 통신을 수반하는 병렬 처리 구조에 적합하다는 이유에서 채택되었다.

2. Actor의 동작

하나의 actor는 메시지 전달로 도착한 통신문(communication)에 의해 다음과 같은 3가지의 동작이 병행 처리되는 연산 대행자(computational agent)라고 정의한다.

① 다른 actor(self 포함)에게 통신문을 전달

② 새로운 행위(behavior)를 갖는 actor로의 변신(replacement)

③ 새로운 actor의 생성(creation)

또, 하나의 actor는 그 actor 이름에 해당하는 우편주소(mail address)와 프로그램의 코드에 해당하는 행위를 가지고 있으며, 우편 주소는 외부에서 들어오는 통신문들을 저장하는 우편 큐(mail queue)를 가지고 있다.

Fig.1에서 보는 바와 같이 actor X_n은 새로운 actor를 생성하거나 연산 처리의 행위만을 제공하는 모듈인 타스크를 생성할 수 있으며, 이러한 생성 처리에 관한 정보를 전달하고 나면 자신은 또 다른 행위를 갖는 또는 새로운 행위를 갖는 actor인 X_{n+1}로 변신한다. 따라서, X_n과 X_{n+1}은 동일한 처리를 가지고 있을 수도 있고, 전혀 다른 처리 방식을 가질 수도 있으며, Fig.1에서의 동작들은 모두 병행적으로 수행된다.

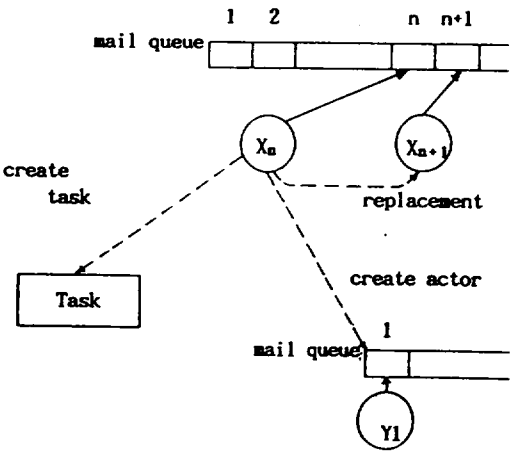


Fig. 1. Behavior of Actor

Actor 모델의 병행성은 각 actor들 사이에, 또는 한 actor의 내부에서 존재하는 데, 이것은 한 actor가 수행될 때 다른 actor들도 동시에 수행될

수 있으며, actor 스스로 동적으로 새로운 actor를 생성하고 생성된 actor에 통신문을 보내 이 actor 역시 동시에 처리되도록 할 수 있다. 또한 하나의 actor 내에서는 각각의 명령문들이 병행 처리된다.

actor 시스템이란 각각의 actor 프로그램을 말하는 것으로 여러 개의 actor들 간에 통신문 전달에 의해서 시스템은 수행되어지며, Actor 시스템은 다음과 같이 정의한다.

o 행위들의 정의

o new 문 : 새로운 actor의 생성

o send 문 : 타스크의 생성

o 수신자(receptionist) actor 선언 : 다른 Actor 시스템에서 통신문을 받을 수 있는 actor들의 목록

o 외부 actor 선언 : 통신문을 보낼 actor들로 현재의 Actor 시스템내에 있지 않은 부분의 actor들의 목록

Actor 모델의 언어로는 SAL(Algol식 표현)과 Act(Lisp식 표현)의 두 종류가 존재하는데, 본 논문에서는 SAL(Simple Actor Language)을 이용하여 설명하려 한다.

예를 들어, n!을 계산하려면 Fig.2의 R-Fact란 행위를 갖는 actor를 new문을 이용하여 생성처리하면 가능하다.

```
def R-Fact(self) [n, cust]
    become R-Fact(self)
    if n=0      then send [1] to cust
        else let c=new R-Cust(n, cust)
            (send [n-1, c] to self)
    fi
end def
def R-Cust(n, cust) [k]
    send [n*k] to cust
end def
```

Fig. 2. SAL program of recursive factorial

R-Fact란 행위를 갖는 actor는 동료(acquaintance) actor로 자신을 가리키며, 계승의 계산을 위한 n값 및 그 결과를 원하는 cust라는 actor를 통신문으로 받아들인다. 통신문이 도착하면 동일한 우편 큐 및 동일한 행위를 갖는 actor로 변신하게 되는데, 이와 병행적으로 if 문장을 수행한다. 즉, 수신한 n 값이 0이면 1을 cust라는 actor에 보내고 일을 완료하며, 0이 아니면 R-Cust라는 행위를 가지고 n과 cust를 동료 actor로 하는 새로운 actor를 new 문장으로 생성하고(여기서는 이를 편의상 c로 부르고 있다), 자신의 우편 큐에 [n-1, c]라는 통신문을 보낸다. 즉, n-1의 계승을 계산하여 그 결과를 방금 생성한 actor인 c에게 보내라는 통신문을 발송하라는 뜻이다. 다음으로 R-Cust라는 행위를 갖는 actor는 동료 actor로 n과 cust라는 actor를 가지게 되며, 통신문 k가 도착하는 동료 actor인 n과 k의 곱을 계산하여 cust에 발송하는 일을 수행한다.

Actor의 순차 합성 (Sequential composition)

1. Actor의 동적 환경

Actor 모델에서는 병행 수행의 기본 모델인 각 actor들 사이의 통신이 동적으로 이루어질 수 있는 환경을 제공하고 있다. 이 동적 환경은 실제로 통신을 수행할 때 그 통신의 출발점을 알 수 있도록 하며, 이는 원시 입력이 계속 변화되는 경우를 지원하기 위한 기법으로 사용된다. 따라서, Actor 모델에서는 각 actor들의 동적 표현을 위해서 사용되는 명령문으로 call g[k]와 같은 형태의 call 구문을 사용하고 있다. 여기서 g는 외부 actor의 우편 주소이며, [k]는 통신문을 의미한다. 즉, 통신문 [k]를 외부 actor인 g에게 전달하고 그 외부 actor g로 부터 실행 결과를 받아 처리하라는 표현이다.

2. 병행 합성의 문제점

기존 프로그래밍 언어들의 대부분은 구문들이

순차적으로 처리되는 순차 합성으로 이루어져 있다. 그러나, Actor 모델에서는 통신문 발송과 새로운 actor의 생성, 변신의 모든 수행이 병행 처리되므로 순차 합성은 사용되지 않는다.

따라서, 일반적으로 표현한 다음과 같은 문장

/예 1/ S1≡x : =x+1

S2≡x : =2*x

에서 병행 합성 S≡S1 ‖ S2의 경우에 S의 결과는 예측할 수 없게 된다('‖'기호는 병행 합성을 의미함). 한 가지 예로서 x=2라고 할 때, S1이 수행되고 S2가 수행되면 그 결과는 6이 되고, S2 다음에 S1이 수행되면 결과는 5가 된다. 즉, 병행 합성에서는 각 문장들이 동시에 처리될 수 있어서 두 문장을 처리하는 순서에 따라 여러 형태의 결과가 발생할 수 있다.

그러므로, 병행 합성으로 각 문장들을 처리한 경우에 그 결과는 다음과 같이 표현할 수 있다.

$$\mathfrak{F}(S1 \| S2)=\mathfrak{F}(S1:S2)\cup\mathfrak{F}(S2:S1)\cup\mathfrak{F}(S1)\cup\mathfrak{F}(S2)$$

여기에서 ':'은 순차 합성의 표기이고, $\mathfrak{F}$ 함수는 $\mathfrak{F}$: D(P)→R(P)와 같이 정의되며, 프로세스의 정의역에서 치역으로 사상되는 함수이다. 프로세스의 정의역은 프로세스가 사용 가능한 변수들의 집합이며, 치역은 프로세스가 변경 가능한 변수들의 집합이다. 이때, S1과 S2가 모두 비분해적(atomic)으로 수행되면 $\mathfrak{F}$(S1)과 $\mathfrak{F}$(S2)는 존재하지 않는다.

그러나, 만일 이 경우에 $\mathfrak{F}$(S1 ; S2)의 한 가지 경우의 결과만 요구된다면 S1과 S2의 처리 순서는 반드시 유지되어야 한다. 그 해결 방법으로 한 프로세서가 전담하여 순차 합성된 S문장을 차례로 처리하게 하면 가능하지만 Actor 모델에서는 모든 처리의 방법이 병행 합성으로만 이루어져 있기 때문에 순차 합성으로 처리한 결과를 원해도 항상 동일한 결과를 보장 받을 수 없게 된다. 다음과 같은 actor 문장을 보자.

/예 2/ S1≡send 〔call g〔cal h〔x〕〕〕 to f

S2≡send 〔call g〔cal h〔y〕〕〕 to f

만일 S1과 S2를 병행 합성으로 수행하게 되면 h라는 actor는 〔x〕나 〔y〕라는 통신문을 받아 다른 행위를 갖는 actor로 변신될 수 있기 때문에 통신문 〔x〕나 〔y〕가 h의 우편 큐에 도착하는 순서에 따라 반환되어지는 결과는 다를 수 있다. 이러한 문제 해결을 위해 병행 합성으로도 순차 합성의 처리를 수행할 수 있도록 고객(customer) actor라는 새로운 actor를 만들어 다음에 처리되어야할 문장을 이 고객 actor에게 전달하여 수행한다. 즉, 이 방법은 하나의 프로세서가 순차적인 문장을 수행하는 것과 동일한 개념으로 해결한 것이다. 이 때, 새로운 고객 actor를 생성하여 처리하는 방식도 세분화하면 다음과 같은 여러 가지 프로세스로 나뉘게 되는데, 이 프로세스들 간에도 병행 처리된다. 그러나, 이들 프로세스 중에서 서로 독립적인 동작을 하는 프로세스들도 순차 합성의 결과를 만족하도록 하기 위해서 하나의 프로세서로만 처리하게 되는데, 이는 병렬성의 정도가 낮아지는 단점을 갖게 된다.

여기서 Fig.3는 기존의 절차 언어에서 처럼 일반적인 순차 처리의 형태를 그림으로 나타낸 것이며, Actor 모델에서는 S1;S2의 순차 합성의 경우 Fig.4와 같이 고객 actor가 또다른 고객 actor를 계속하여 생성하는 계층 처리 방식으로 처리함으로써 해결한다.

Fig.3처럼 순차 기계(단일 프로세서로 처리하는 기계)로 처리하는 것과 같이 각 actor가 처리되도록 하면 순차 합성에 따른 문제점을 해결할 수 있지만 기본적으로 각 actor는 병행 합성이므로 그 처리 순서의 제어가 어렵다. 따라서, Fig.4처럼 고객 actor를 이용하여 단계적으로 처리의 권한을 위임하는 delegation 기법으로 처리하면 actor의

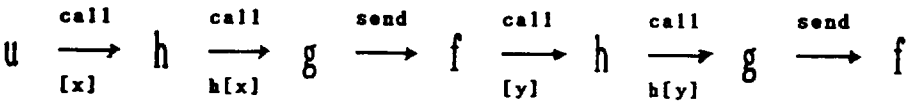


Fig. 3. sequential composition of sequential machine

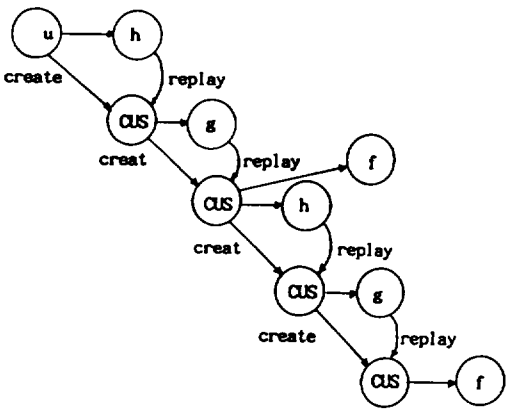


Fig. 4. processing mechanism of /example 3/

병행 합성 방법만으로도 해결이 가능하다. 그러나, 앞에서 언급한 바와 같이 병렬성의 정도가 낮아지게 되는 문제점(다시 말하면, 순차도가 증가하는)을 내포하고 있기 때문에 다중 프로세서 구조로 된 시스템의 성능을 최대로 하기 위해서는 동일한 처리 결과를 유지하면서 최대 병렬성 및 최소 순차도를 보장해 주는 것이 필요하다.

3. Actor의 순차 합성

Actor 시스템에서 문장의 순차 합성을 표현하는 방법은 ';'으로 표기한다. 그러나 Actor의 커널 언어에서는 순차 합성에 대한 어떠한 표현도 없기 때문에 순차 합성으로 표현된 프로그램은 고객 actor를 생성·이용하여 처리하고 있다. Actor 모델에서는 앞의 Fig.4와 같이 순차 합성의 처리 순서를 해결하고 있다.

Actor 모델에서 순차 합성의 경우 문제가 발생되는 것은 동적 처리 환경의 개념 때문에 발생하는데, 하나의 외부 actor를 여러번 호출(call)했을 때, 이 외부 actor의 처리 행위는 전달된 통신문에 의해 변신될 수 있다. 따라서, 순서가 유지되면서 처리되야만 하는 call문의 경우에는 기존 Actor 모델에서의 고객 actor 처리 방식은 병렬성을 감소시킨다. 또한, 이 call 구문을 여러 문장에서 사용할 수 있지만 병렬성의 문제가 발생하는 것은 send 문장내에서의 call 문을 순차 합성으로 사용한(/예2/) 경우이다.

그 이유는 처리 행위의 정의에서는 배정문이 존재하지 않으며, 새로운 actor의 생성은 상호 의존성이 없기 때문이다. 또한, 선언문은 프로세서의 처리 대상에서 제외되기 때문이다. 아울러, Actor 모델에서는 send 문장에 수식(expression)을 사용할 수 있도록 허용하고 있는 것은 재귀적 구조를 지원하기 위해서며, SAL의 커널 구현에 있어서 고객 actor의 생성 처리는 사용자에게는 감춰지는 부분이므로 편리성을 제공할 수 있다는 이유에서이다. 따라서, send 문의 수식을 순차 합성 처리시 의존성에 의한 고려 사항은 다음 4가지로 정리할 수 있다.

① 산술 연산

② 호출(call) 연산 : 다른 actor의 호출

③ 지연(delay) 처리 연산 : 평가 명령이 있을 때까지 처리를 지연. 즉, actor의 우편 주소만 유지하고 있음.

④ 목표(target) actor : 결과를 응답할 목표 actor

최대 병렬성 보장을 위한 알고리즘

앞에서 기술한 예와 같은 순차 합성의 경우에 다음과 같은 정리를 이용하면 원래의 시스템과 수행한 후의 결과가 항상 동일한 최대 병렬 처리 시스템(Maximally Parallel System;MPS)을 생성할 수 있다.

〔정의 1〕 주어진 관계를 만족하면 수행 순서에 관계없이 결과가 동일한 시스템을 결정적 (deteminate) 시스템이라 한다.〔Maekawa87〕

〔정리 1〕 프로세스들간의 상호간섭이 없는 시스템을 결정적 시스템이다.〔Maekawa87〕

〔정리 2〕 프로세스들간의 처리 순서에 대한 선행 관계(precedence realtion)⇨를 갖는 프로세스들의 결정적 시스템이 주어졌을 때, 같은 프로세스들 간의 새로운 선행관계시스템⇨'를 정의할 수 있다면 새로운 시스템은 유일하며, 최초의 시스템과 이 새로운 시스템인 MPS는 수행한 후 항상 같은 결과를 생성하는 동치 시스템이다.

Meakawa87]

$\Rightarrow' \equiv \{(P_i, P_j) \in c1(\Rightarrow) \mid (R(P_i) \cap R(P_j)) \cup (R(P_i) \cap D(P_j)) \cup (D(P_i) \cap R(P_j)) \neq \phi\}$

(단, P_i : 프로세스, D : 정의역, R : 치역, c1 : 이행성 폐포(transitive closure)를 의미하며 프로세스의 정의역과 치역은 actor의 우편 주소들의 집합임)

Actor 모델에서 존재하는 순차성은 결국 통신문큐(mail queue)에 통신문(communication)이 도착해야 처리할 수 있다는 것뿐이므로 위의 정리를 이용하여 Actor 모델에서 순차 합성 문장을 탐색하고 같은 그룹으로 묶은 다음, 아래와 같은 알고리즘에 의해 그 병렬성을 극대화시킬 수 있다. 여기에서 두 가지 알고리즘을 제시하는데, 〔알고리즘 1〕은 위의 두 정리에 근거하여 Actor 모델의 해(MPS)를 구하는 방법이고, 〔알고리즘 2〕는 병렬성 향상을 위해 순차적 처리를 최소화시키는 것에 촛점을 맞추어 생성되는 고객 actor가 단순히 다른 actor에게 통신문을 전달(forwarding)하는 기능만을 갖으면 이를 제거하여 프로세스의 수를 줄이고, 순차 처리 관계에 있는 프로세스들 사이의 선행 관계를 재조정하여 최소 순차도를 갖게 하는 알고리즘이다. 즉, 이 방법은 이러한 고객 actor를 제거함으로써 프로세스의 수를 줄이고 최소 순차도를 보장하도록 하는 것이다.

다음의 정의들은 최종적으로 제시된 처리 방법을 비교하는데 사용된다.

〔정의 2〕 시스템의 순차도 : 프로세스를 node로 하는 선행 시스템에서 최장 경로상의 node의 수

〔정의 3〕 시스템의 병렬도 : 프로세스를 node로 하는 선행 시스템에서 어느 한 순간 동시에 수행될 수 있는 node들 수의 최대값

〔알고리즘 1〕 : 〔정리 1〕을 직접 Actor 시스템에 적용하는 방법

이 알고리즘은 각 프로세스의 처리 순서를 중심으로 프로세스간의 링크를 생성하고 생성된 링크 중에서 이행성 폐포가 발생하는 링크를 제거하는 방법이다.

Step 1 send문 내의 수식을 후위 표기(postfix)로 표현하는데, 이는 actor 문장에서 프로세스을 추출할 수 있도록 하기 위한 것이다.

-의존도 검사를 위한 기호 테이블(symbol table)의 생성

Step 2 후위 표기 내의 call 문장을 분해하여 다음과 같은 프로세스들로 나누며, 이들 각각의 프로세스가 하나의 actor로 활동하게 된다.

i) 단순 호출의 경우

call g(x)는 ①actor g 호출, ②g의 고객 actor 생성, ③g에서 고객 actor로 결과를 응답하는 프로세스들로 분해된다.

ii) 중첩된 호출의 경우

call g(call h(x))는 ①actor h 호출, ②h의 고객 actor 생성, ③h에서 고객 actor로 결과를 응답, ④h의 고객 actor에서 g 호출, ⑤g의 고객 actor 생성, ⑥g에서 고객 actor로 응답하는 프로세스들로 분해된다.

이 때, 새롭게 생성되는 고객 actor도 독립된 actor이므로 유일한 우편 주소를 가져야 한다. 이 유일성을 부여하기 위해서 우편 주소의 이름으로 점(dot) 표기 방법을 사용한다.

(예) C1, C1.1, C1.1.1, …

-분해된 call 문장에서 나타나는 기호들도 기호 테이블에 저장 관리한다.

Step 3 기호 테이블을 이용하여 각 프로세스들의 정의역과 치역을 결정하는 의존도 테이블을 생성한다. 이 과정은 〔정리 2〕를 사용하기 위해 프로세스의 정의역과 치역에 해당하는 actor를 결정하는 것이다.

Step 4 〔정리 2〕에 의해서 두 프로세스의 정의역과 치역의 의존도를 검사하여 〔그림 10〕에서와 같이 의존성이 발견된 프로세스를 가지고 프로세스 관계 행렬의 해당 원소를 true(T)로 지정한다. 이 과정에서 고려중인 n개의 문장을 처리하는데 필요한 모든 프로세스의 수를 p라고 하면 각 프로세스의 의존도를 검사하기 위한 시간 복잡도는 한 프로세스에 대하여 모든 프로세스와 비교되므로 $O(p)$라 할 수 있다.

Step 5 프로세스 관계 행렬에서 하나의 프로세스에 대해서 발생되는 이행성 폐포항을 소거한다. 이 이행성 폐포항을 제거하는 방법은 우선 한 프로세스 P_i에 대해 P_i행에서 그와 인접해 있는 프로세스 P_j를 찾고 이 P_j와 다시 인접해 있는 프로

세스 P_k를 찾는다. 이렇게 찾아진 프로세스의 링크인 (P_j,P_k)가 P_j행에서 P_k로 가는 의존도가 true이면 이 P_k는 이행성 폐포항으로 제거한다. 이때 프로세스의 수를 p라 하면 각 프로세스에 대해 인접된 링크를 찾기 위한 시간 복잡도는 $O(p^2)$이며, 이에 대해 다시 P_j행에서 P_k로 가는 링크의 존재 여부를 위한 복잡도가 $O(p)$이므로 전체적인 복잡도는 $O(p^3)$이다. 이때 Actor 모델에서의 프로세스 수 p는 n개의 모든 문장이 갖는 actor의 연산자 수와 같다.

Step 6 생성된 MPS의 순서쌍을 구현한다.

〔알고리즘 2〕: 전달(forwarding) 기능의 고객 actor를 제거하여 최소 순차도를 갖도록 하는 알고리즘

만일 $S_1;S_2;\cdots S_n$의 send문에 의한 순차 합성이 주어졌을 때, Actor 모델에서 최소 순차도를 보장할 수 있는 알고리즘은 다음과 같다.

Step 1. 문장 $S_1,S_2\cdots,S_n$ 각각에 대해 다음을 행한다.

Step 1.1 send 문장 내의 수식을 후위 표기 또는 전치 표기(prefix)로 표현하고 선행 관계를 고려한 프로세스들의 선행 관계 그래프를 작성한다. 이때 각 call 문장마다 1개의 customer가 생성되는데, 최후에 생성되는 customer가 다른 actor에게 전달하는 기능만을 가지면 customer actor를 생성하지 않고 최종 함수 actor가 직접 target actor에 송신하게 한다.

Step 1.2 customer들을 포함하는 모든 actor를 그 주소와 함께 symbol table에 저장한다.

Step 1.3 문장내에서 호출된 각 함수에 대해 문장 번호, 프로세스 명칭과 함께 function table에 저장한다.

Step 1.4 target actor에 대해 문장 번호, 프로세스명을 function table에 저장한다.

Step 1.5 Step 1.1과 Step 1.2에서의 결과를 이용하여 프로세스들의 정의역과 치역을 dependency table에 저장한다.

Step 2. function table에 저장된 각각의 함수(여기서는 fun이라고 가정한다)에 대해 다음을 수행한다.

Step 2.1 fun을 호출한 프로세스들을 순서대로 $P_1,P_2,\cdots,P_n$이라 하면 다음과 같은 알고리즘에 의해 최소 순차도를 얻을 수 있다.

```
〔선행 그래프를 처리하는 알고리즘〕
  if n>1 then
      for i := 1 to n-1 do
          edge(P_i,P_{i+1})을 선행 관계 그래프에
          추가한다.
```

Step 2.1에서 fun을 호출한 프로세스를 선행 관계 그래프에 추가하는 것은 변신에 의한 순차성을 부여하기 위함이다. 여기서, 선행 그래프는 순서쌍들의 집합으로 구현할 수 있으며, 다른 함수로 변신하지 않는 함수를 미리 알 수 있다면 그러한 함수에 대해서는 Step 2.1을 생략할 수 있다.

〔알고리즘 2〕는 프로세스들 간의 링크를 직접 그래프에 추가하는 방법으로서 복잡도는 프로세스의 수 p에만 관계한다. 따라서, 이 프로세스들을 후위 표기 시간 및 선행 관계 그래프에 추가하는 행위에 대한 시간만 계산하면 되므로 복잡도는 $O(p)$이다.

알고리즘의 실행 예 및 비교

1. 실행 예

위의 두 가지 알고리즘을 사용하기 위해서 편의상 다음과 같이 기호들의 표기를 정의하였으며, /예 2/에 표현된 순차 합성의 경우를 실행해 본 결과이다. 여기서 사용된 기호는 다음과 같이 정의한다.

^ : actor의 호출

$: 고객 actor의 생성

@ : 고객 actor로의 응답

: 목표 actor로 전달

C : 고객 actor의 우편 주소(각 고객 actor의 우편 주소는 유일함)

o /예 2/에 표현된 두 문장 S1, S2를 후위 표기를 이용하여 표기한 프로세스의 실행 순서는 다음과 같다.

S1 → xh^ C$ C@ g^ C$ C@ f#
process# → Ⓐ Ⓑ Ⓒ Ⓓ Ⓔ Ⓕ Ⓖ

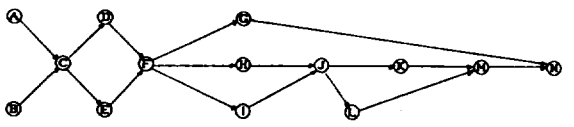
S2 → yh^ C$ C@ g^ C$ C@ f#
process# → Ⓗ Ⓘ Ⓙ Ⓚ Ⓛ Ⓜ Ⓝ

No.	symbol	attribute
1	x	mail address of actor
2	h	mail address of actor
3	c1	mail address of customer
4	g	mail address of actor
5	c2	mail address of customer
6	f	mail address ofactor
7	y	mail address of actor
8	c3	mail address of customer
9	c4	mail address of customer

Fig. 5. Symbol Table

process#	domain	range
Ⓐ	1, 2	2
Ⓑ	3	3
Ⓒ	2, 3	3
Ⓓ	3, 4	4
Ⓔ	5	5
Ⓕ	4, 5	5
Ⓖ	5	6
Ⓗ	2, 7	2
Ⓘ	8	8
Ⓙ	2, 8	8
Ⓚ	4, 8	4
Ⓛ	9	9
Ⓜ	4, 9	9
Ⓝ	9	6

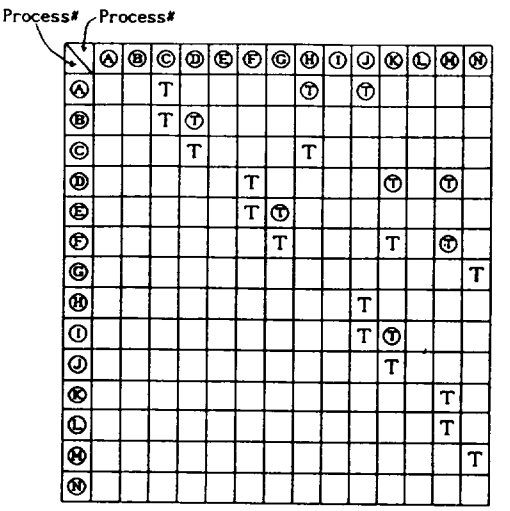
Fig. 6. Dependency Table



(parallelity: 3, sequentiality: 9, no. of process: 14)

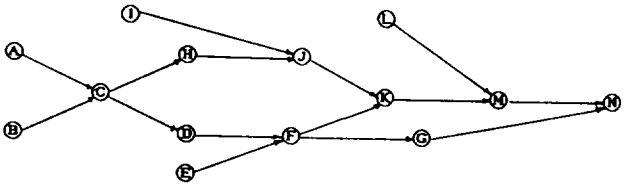
Fig. 7. sequential composition graph of basic actor model

2. 알고리즘 1로 실행한 경우



여기서 Ⓣ는 이행성 폐포항으로 제거되고 T만 남는다.

Fig. 8. relation matrix of processes



(parallelity: 5, sequentiality: 7, no. of process: 14)

Fig. 9. sequential composition graph of MPS

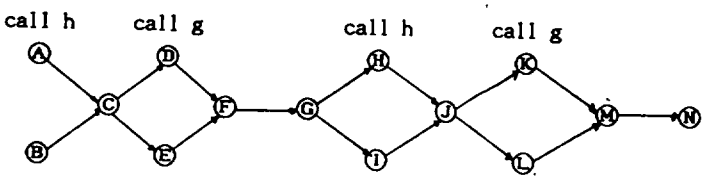
3. 알고리즘 2로 실행한 경우

process#	domain	range
Ⓐ'	1, 2	2
Ⓑ	3	3
Ⓒ	2, 3	3
Ⓓ'	3, 4, 5	4, 5
Ⓖ	5, 6	6
Ⓗ	2, 7	2
Ⓘ	8	8
Ⓙ	2, 8	8
Ⓚ'	4, 8, 9	4, 9
Ⓝ	9, 6	6

Fig. 10. Dependency Table

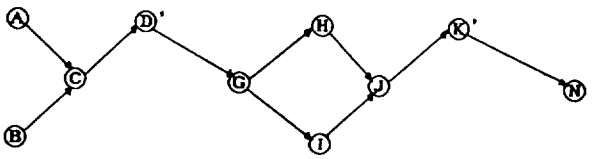
function	process#	sentence
f	G	S1
f	N	S2
g	D'	S1
g	K'	S2
h	A	S1
h	H	S2

Fig. 11. Function Table



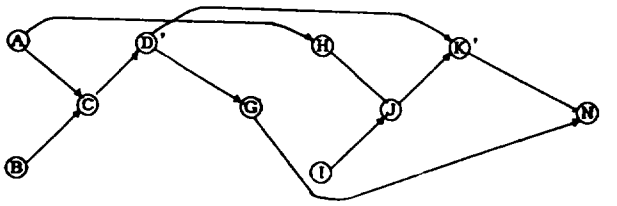
(parallelity: 2, sequentiality: 10, no. of process: 14)

Fig. 12. Using customer Method to call statement in Basic Actor Model



(parallelity: 2, sequentiality: 8, no. of process: 10)

Fig. 13. Graph of Eleiminated Forwarding Customer



(parallelity: 3, sequentiality: 5, no. of process: 10)

Fig. 14. Sequential Composition Graph of Minimally sequentiality

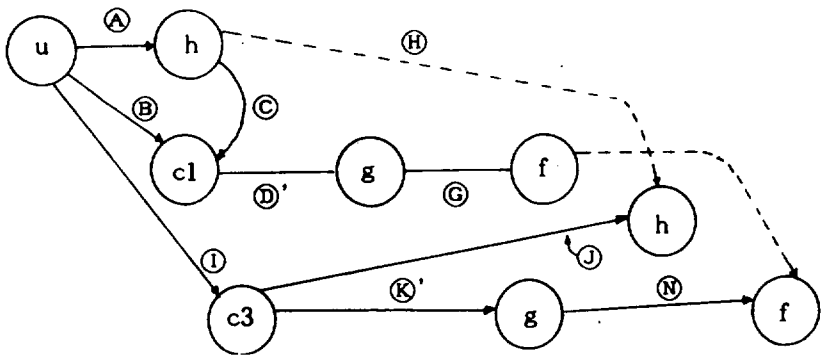


Fig. 15. Processing Method of Sequential Composition Graph of Minimally sequentiality

4. 알고리즘의 비교

〔알고리즘 1〕의 특성은 최대 병렬성 보장을 위한 알고리즘의 정리에 의한 것으로 의존도가 존재하는 프로세스를 모두 사상시켜 순서쌍을 일차적으로 생성하고, 생성된 순서쌍들 간에 이행성 폐포가 발생하는 순서쌍을 제거하는 두 단계로 이루어져 있다. 따라서, 이행성 폐포 관계항의 발생 여부를 검사하고 제거하는 작업이 부수적으로 요구된다.

〔알고리즘 2〕는 최소 순차도를 보장함으로써 최대 병렬성을 지원할 수 있다는 점에 착안하여 행렬 연산에 의존하지 않고 선행 관계만을 검사해서 그래프 형식으로 처리하는 알고리즘이다. 즉, 순차 합성을 위한 기본 처리 그래프에서 전달 기능만을 갖는 고객 actor를 생성하는 프로세스를 제거한 후, 파악된 프로세스의 정의역과 치역에 대한 의존도를 검사하여 선행 관계 그래프에 추가시키며 최소 순차도를 보장하는 방법이다.

여기서 제시한 각 알고리즘들을 평가하기 위해서 고려되어야 할 사항은 call 문장의 내포 정도(degree of nesting), 산술식 길이(length's arithmetic expression) 그리고 순차 합성을 행하려는 문장들의 수 등이다. 따라서 제시된 알고리즘들의 시간에 대한 복잡도를 평가할 때 순수한 알고리즘에 나타나는 복잡도는 앞장에서 논한 바와 같이 〔알고리즘 1〕은 $O(p^3)$으로 평가되었고, 〔알고리즘 2〕의 복잡도는 $O(p)$로써 복잡도를 평가한 결과로는 가장 효율적인 것으로 나타났으며, 앞의 실행 예에서 보인 그래프에서도 현저하게 순차도가 감소함을 알 수 있었다.

〔알고리즘 2〕는 프로세스의 수를 줄임으로써 순차도를 최소로 하는 것을 목표로 하다. 그러므로, 이 경우에서는 순차 합성 문장의 수를 n이라고 하면 n에 따라 전달 기능만 갖는 고객 actor를 n개만큼 제거할 수 있다. 따라서 〔알고리즘 2〕는 순차 합성 문장의 개수가 n일 때, 프로세스의 수 p는 2n만큼 줄어든다. 이와 같이 순차 합성의 처리가 요구되는 문장들이 다단계의 문장들로 구성된 순차 합성의 경우 〔알고리즘 2〕로 처리하면 프로세스의 수와 순차도가 급격히 감소됨을 알 수 있었으며 그 결과는 Fig.16과 같다.

여기서 ①의 경우는 순차 합성 문장의 수가 n인 경우 생성되는 프로세스의 수를 비교한 것으로서 한 문장당 생성되는 프로세스의 수를 P_i라고 할 때, 전체 문장들에서 생성되는 프로세스의 수는 $n \cdot \max(P_1, P_2, \cdots, P_n)$을 넘지는 못하는데 〔알고리즘 2〕의 경우는 전달 기능의 고객 actor를 제거함으로써 한 문장당 customer를 생성하는 프로세스와 target actor로 전달하는 프로세스의 2개 프로세스가 줄어들게 된다.

②의 경우는 알고리즘을 처리하기 위한 행렬의 크기를 비교한 것으로서 〔알고리즘 1〕은 프로세스 관계 행렬에 의해 p^2이고, 〔알고리즘 2〕는 행렬을 사용하지 않으므로 0이다.

Factor	[Algorithm 1]	[Algorithm 2]
① if SC#=n, number of creating processes	$(p_1+p_2+\cdots+p_n) \le n\cdot\max(p_1,\cdots,p_n)$	$(p_1-2)+\cdots+(p_n-2) = (p_1+\cdots+p_n)-2n \le n\cdot\max(p_i)-2n$
② Matrix Size for Algorithm	p^2	Unused Matrix
③ Number of increasing processes for creating customer by nested degree of call functions	$2c$ (c: nested degree of call function)	*process# is reduced by elimination of forwarding Actor*
④ Processing time of each algorithms to number of process p	$p * p * p$	$s + p + f < 2p$
⑤ time complexity	$O(p^3)$	$O(p)$

SC#: number of sequential composition statements

n : number of sequential composition statements,

p_i : number of processes for processing i_{th} statement ,

$p=\Sigma p_i$; number of all processes for processing sequential composition statements n,

s : number of all symbols to used n statements,

f : number of functions to stored function table,

c : nested degree of call functions

Fig. 16. Performance of Algorithms by each factors

③은 한 문장에서 call 함수의 내포 정도에 따라 생성되는 customer에 관련된 프로세스의 수로서 call 함수가 내포될 때마다 customer는 새롭게 하나씩 생성해야 하고 응답해주어야 하므로 call 함수의 내포 정도 c에 2를 곱한 결과가 된다.

④의 경우는 알고리즘을 처리하기 위한 최대 시간을 나타낸 것으로 〔알고리즘 1〕은 이행성 폐포항의 제거를 위해 소요되는 시간으로 인해 p*p*p 이고, 〔알고리즘 2〕는 링크로 처리하므로 하나의 프로세스에서 테이블에 의해 다음 프로세스를 연결만 지어주면 되므로 s와 함수 테이블에 저장된 함수의 수 f, 그리고 프로세스의 수 p를 더한 값과 같다. 여기서 s, f, p의 관계는 s+f≤p로서 s+p+f≤p인 관계가 성립한다.

⑤의 시간에 대한 복잡도는 앞 절에서 설명한 바와 같으며, 이상에서 살펴본 바와 같이 최소의 순차도를 갖도록 하여 병렬성을 향상시키기 위해 〔알고리즘 2〕가 매우 효율적인 것으로 평가되었다.

적 요

Actor 모델은 병렬 처리 구조를 기반으로 하는 병행 합성 방법을 사용하고 있기 때문에 특별히 문장의 처리 순서가 유지되어야 하는 순차 합성의 경우에는 고객 actor를 생성하여 이용하는 방법으로 처리하고 있다. 그러나, 이 고객 actor로 하여금 다음 문장의 처리를 전가시키는 방법에서 발생되는 병렬성 감소로 인하여 Actor 모델 자체의 미세화 병행성 개념에 위배된다.

따라서, 본 논문에서는 이를 개선하기 위한 방법으로서 순차 합성의 경우에 각 프로세스들 간에 의존성을 평가하여 동일한 시스템을 유지하면서 병렬성을 극대화하는 MPS 알고리즘을 세 가지로 제시하였다. 제안된 알고리즘을 이용한 MPS는 앞장에서 살펴본 바와 같이 그 병렬성은 기존의 Actor 모델에서와 같이 고객 actor를 이용하는 처리 방식보다 증가되었으며, 또한 〔정리 1,2〕를 이용한 알고리즘보다는 전달 기능의 고객 actor를 제거하는 방법이 궁극적으로 프로세스의 수를 줄일 수 있기때문에 매우 효율적인 방법으로 평가되었다.

이 알고리즘은 Actor 모델의 구현을 위한 커널의 설계에 있어서 병렬 처리 구조가 갖는 병행 처리의 정도를 최대로 부여하기 위해 제안하였다.

참 고 문 헌

Agha, G., 1986. Actors : A Model for Concurrent Computations in Distributed Systems, MIT Press.

America, P., 1987. POOL-T : A Parallel Object-Oriented Language, Yonezawa A. & Tokoro M., MIT Press.

Athas, W. and C., Seitz, 1988. Multicomputers : Message-Passing Concurrent Computers, IEEE Computer.

Bronnenberg, W. and L., Nijman, 1987. DOOM : A Decentralized Object-Oriented Machine, IEEE Software.

Cox, B., 1986. Object-Oriented Programming : An Evoluationary Approach, Addison-Wesley.

Dally, W. and L., Chao 1987. Architecture of a Message-Driven Processor. ACM Computer Architecture Conference.

Harland, D., 1986. Currency and Programming Languages, Halsted Press.

Ishikawa Y. and M., Tokoro 1987. Oriented 84/K : An Object-Oriented Concurrernt Programming Language for Knowledge Representation, MIT Press.

Kronsjö, L., 1979. Algorithms : Their Complexity and Efficiency, Wiley.

Lieberman, H., 1981. A Preview of Act-1, AI-Memo 625, Artificial Intelligence Laboratory, MIT.

Liskov, B. and R., Scheifler 1983. Guardians and Actions Lingustic Support for Robust Distributed Systems, TOPLAS.

Maekawa and Oldehoeft, 1987. Operating Systems : Advanced Concepts, Benjamin/Cummings Publishing Co.

Moss, E., 1987. Panel Discussion : Object-Oriented Concurrency, OOPSLA'87 Addendum to the Proceedings.

Mullender, S., 1985. Principles of Distributed Operating System Design, Mathemetisch Centrum.

Refenes, A., 1988. N-Expression Implementations for Integrated Symbolic & Numeric Processing, Future Generation Computer Systems.

Yonezawa, A. and M.Tokoro, 1986. Object-Oriented Concurrent Programming, MIT Press.

원유헌, 1991. 프로그래밍 언어론, 정익사.

원유헌, 양승의, 이기철, 곽호영, 1990. 객체 중심 언어에서의 소프트웨어 구성요소 공유 기법, 한국정보과학회 춘계학술발표논문집, Vol.17, No.1, 355~358.

이기철, 곽호영, 원유헌, 1990. Actor 시스템의 순차 합성법을 위한 병렬 처리의 극대화, 세계한민족 과학자 종합학술대회 논문집.