시뮬레이션의 계층적 애니메이션

Hierarchical Animation for Simulation

이미라* · 조대호*
Yi, Mira and Cho, Taeho

Abstract

There are many issues in computer simulation such as verifying model code, validating models, understanding the dynamics of systems and training the personnel. The developers of simulation tool have been interested in the animation since it can help solve the problems related to the above listed issues. In practice, animation is one of the popular method for displaying the simulation output for solving these problems. Trying to display all the graphic objects representing the dynamics of the models being simulated, however, causes the distraction of focus, which results in solving the above listed problems difficult. The redundant graphic objects also increase the computer computation overhead. This paper presents a hierarchical animation environment in which the users can have better focus on the dynamics of system components. In hierarchical animation environment the users can observe the dynamics of system by selectively choosing the hierarchical level and components with in a level of the hierarchically structured model. Especially when the model is large and complex the selection of observation level is needed. The design approach of the hierarchical animator is based on the DEVS(Discrete EVent system Specification) formalism which is theoretically well grounded means of expressing modular and hierarchical models.

Key Words: 계층적 애니메이션, DEVS, 애니메이터, 애니메이션의 재사용성.

* 성균관대학교 전기전자 및 컴퓨터 공학부

1. 서론

컴퓨터 애니메이션은 관심의 대상이 되는 시스템의 동적인 특징(dynamics)을 움직이는 그래픽 객체로 보이게 하는 출력의 한 형태로써 관찰자가 관심의 대상이 되는 객체를 직관적으로 인식하게 하여 시스템을 쉽게 이해하도록 한다[13]. 이러한 장점 때문에 여러 컴퓨터 응용 분야에서 애니메이션을 출력 형태로 하고 있으며, 시뮬레이션은 그 중 한 분야이다.

시뮬레이션에서는 신뢰성 있는 시뮬레이션을 실현하기 위해 개발자가 수행하는 모델코드에 대한 검증, 개발자와 시스템 전문가(시뮬레이션 요구자) 사이의 모델 검증, 시뮬레이션 관찰자 입장에서 시스템의 동적인 변화를 이해하는 정도, 개별적인 시뮬레이션의 훈련 등이 논제로 다뤄지고 있다[1,3]. 시뮬레이션 툴 개발자들은 언급한 논제로 인해 발생되는 문제점들을 개선하고자 더 나은 툴 개발을 연구하고, 이의 해결 방법의 하나로 시뮬레이션 과정을 애니메이션으로 출력하는 것에 관심을 갖고 있다[14].

시뮬레이션이 시스템에 대한 변화를 예측 또는 평가하기 위한 것인 만큼, 그 수행 과정 및 결과를 애니메이션으로 표현하여 이해하기 쉽도록 하는 것은 중요하다[1,14]. 시뮬레이션 이해의 용이성은 시뮬레이션에 익숙하지 않은 시스템 전문가가 시뮬레이션 모델링 과정에 참여 할 수 있도록 유도하여 시뮬레이션 모델에 대한 신뢰도를 높이는 역할을 하고, 시뮬레이션 개발자 측면에서는 통계적인 수치 결과로는 파악하기 어려운 시뮬레이션 진행과정을 직관적으로 이해함으로써 모델에 대한 검증을 좀 더 쉽게 하도록 한다[1,7]. 실제로 상용화된 많은 시뮬레이션 개발 환경들이 애니메이션 환경을 제공하고 있으며 이러한 소프트웨어에는 AutoMod, MODSIM III, Arena, Factor-AIM, SIMAN 등이 있다[2].

시뮬레이션이 현실적으로 수행하기 어려운 시스템을 모의 실험한다는 특성상 대상 시스템은 크고 복잡한 경우가 많으며, 이러한 시스템의 변화를 관찰하고자 한다면 시뮬레이션 모델구조와 애니메이션 또한 그와 비슷한 복잡도로 행해져야 할 것이다. 그러나, 모든 시스템 구성 요소들을 한꺼번에 애니메이션 하는 일은 그 복잡도 때문에 관찰하고자 하는 시스템 영역에 대한 초점을 흐리게 할 수 있다. 또, 시뮬레이션이 많은 컴퓨팅 자원을 요구하는 일인 것을 고려할 때, 복잡한 애니메이션 처리는 시스템에게 더욱더 많은 부하를 줄 것이다.

본 논문에서는 이러한 문제들을 개선하고자 전체 시스템에서 관찰하고자 하는 계층을 사용자가 선택하여 애니메이션 하는 계층적 애니메이션을 제안하고, 계층적 애니메이션을 하기 위한 프로세서인 계층적 애니메이터를 설계하고 검증하는 것을 핵심 내용으로 한다. 제 2장에서는 계층적 애니메이션을 하기 위한 배경 이론인 DEVS 형식론과 시뮬레이션 애니메이션에 대한 관련 연구를 소개하고, 3장에서는 계층적 애니메이션을 하는 시뮬레이션의 전체적인 개발 환경을 설계하고, 애니메이션의 스케줄링 프로세서인 계층적 애니메이터 설계를 4장의 내용으로 한다. 제 5장에서는 DEVS-C++ 시뮬레이터를 이용한 구현을 통해 설계 내용을 검증하고, 6장에서 결론 및 향후과제를 제시한다.

2. 배경이론 및 관련 연구

시뮬레이션에 있어서 모델이란 실제(또는 가상) 시스템을 시뮬레이션 목적에 맞게 단순화 및 추상화 시켜 표현한 것을 말하며, 이러한 과정을 모델링이라 한다[2,5]. DEVS(Discrete EVent system Specification) 형식론은 연속적인 시간상에서 이산사건을 발생시키는 시스템을 시뮬레이션하기 위해 이론적으로 정립된 모델링 방법론이다[2]. 이는 모델의 구조와 행동을 시뮬레이션 수행으로부터 추상화시키기 위해 모델을 집합 이론적 방법을 이용한 것으로, 시스템을 계층적(hierarchical)이고 모듈화(modular) 된 형식으로 기술한다[2,12,17]. 이러한 특징은 복잡한 시스템에 대해 보다 신뢰성 있는 모델링을 가능하게 하는 장점이 있어 계층적 시뮬레이션 애니메이션을

하기 위한 이론적인 토대가 된다.

2.1절에서는 DEVS 형식론에 대해 서술하고, 2.2에서는 시뮬레이션과 애니메이션의 연결에 관한 연구들을 소개한다.

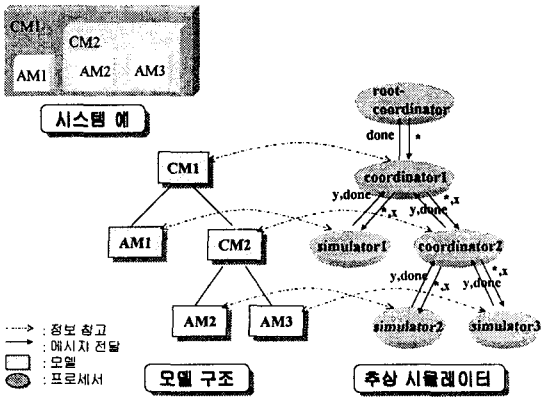
2.1 DEVS 형식론

DEVS 형식론에서 시스템을 기술하기 위해 제안한 모델 형태에는 기본(basic)모델, 결합(coupled)모델이 있다[2]. 기본 모델은 시스템의 최하위 구성 요소들을 표현하기 위한 것이고, 결합 모델은 시스템의 구성요소 간에 상호작용을 표현하기 위한 것이다. 결합모델은 다른 상위 계층의 결합모델을 위한 구성 모델로 사용될 수 있기 때문에 모델이 완성되면 시스템은 기본 모델과 결합모델로 구성된 하나의 결합모델이 최상위 노드에 있는 트리 형태로 표현 될 수 있으며, 시스템과 유사한 계층성을 모델구조로 갖게 된다.

$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, t_a >$
- X : 입력 이벤트의 집합
- S : 상태집합
- δ_{int} : 내부 상태변이 함수 - S -> S
- δ_{ext} : 외부 상태변이 함수 - Q * X -> S
- λ : 출력 함수
- t_a : 시간 갱신 함수

$DN = < D, \{M_I\}, \{I_I\}, \{Z_{I,j}\}, select >$
- D : 구성요소 이름
- M_I : 구성모델
- I_I : 모델i와 연관된 모델의 집합
- $Z_{I,j}$: 모델 i와 j모델간의 연결함수
- select : tie-breaking selection 함수

DEVS 형식론을 기반으로 만들어진 시뮬레이션 환경들(DEVS-Scheme, DEVS-C++,....)은 모델을 이벤트 스케줄링하기 위한 프로세서(이후 시뮬레이터)를 제공한다. 이들은 DEVS 모델의 모듈화 특성에 맞게 객체 지향적인 시뮬레이션이 가능하도록 모델 클래스와 시뮬레이터 클래스가 정의되어 있지만, 환경에 따라 모델과 시뮬레이터가 하나의 클래스로 묶여있는 것도 있다. 이러한 환경들은 모델을 정의하는데 있어서는 DEVS 이론을 따르므로 차이가 없으나, 시뮬레이터가 이벤트 스케줄링을 하는데는 다소 차이가 있다. 본 논문에서는 시뮬레이터의 모듈화가 잘 되어있어 추상 시뮬레이터 구조가 모델 구조와 같은 시뮬레이션 환경인 DEVS-Scheme을 기준으로 용어 및 관련 개념을 설명한다[2].



<그림 1> 모델 구조와 추상 시뮬레이터 구조

우선 시뮬레이터 종류를 살펴보면 기본 모델, 결합모델에 해당하는 simulator, coordinator가 있고, 전체 시뮬레이션 시간을 제어하기 위한 root-coordinator가 있다. 즉, 모델마다 simulator 또는 coordinator와 쌍을 이루고, 모델 구조에 단 하나의 root-coordinator를 갖는다. <그림 1>은 간단한 시뮬레이션 대상 시스템, 모델구조, 추상 시뮬레이터 구조를 트리 형태로 나타내고 그들간의 상호연관성을 표현한 것이다. 그림에서의 추상 시뮬레이터는 simulator, coordinator, root-coordinator가 모델 구조에 맞게 결합하여 전체 시스템을 위한 하나의 시뮬레이터 역할을 하는 것을 의미한다.

시뮬레이션 실행은 추상 시뮬레이터 구조를 이루는 시뮬레이터들의 상호 메시지 전달을 통한 이벤트 스케줄링에 의해 이루어진다. Simulator와 coordinator는 모델의 상태변이(transition)가 일어날 때마다 상위 프로세서에게 다음 이벤트가 일어날 시간 정보를 done-메시지에 실어 올려보낸다. 마지막으로 done-메시지를 받은 root-coordinator는 해당 시간만큼 시뮬레이션 시간 값

을 갱신하고 다음 이벤트에 대한 스케줄링 시작을 *-메시지를 보냄으로써 지시하게 된다. Simulator에서 나오는 출력은 상위 coordinator에게 y-메시지에 실려 전달되고, coordinator는 이 출력정보를 x-메시지에 실어 연관된 다른 프로세서에게 전달한다. 즉, y-메시지를 받았을 때 외부 상태변이(external transition)를, *-메시지를 받았을 때 내부 상태변이(internal transition)를 일으킨다. 이렇게 시뮬레이션은 시뮬레이터들 간 4개의 메시지들을 주고받으며 진행된다.

메시지와 함께 이벤트 스케줄링을 하는데 필요한 요소로는 시뮬레이션 시간 값을 나타내는 변수 t_L, t_N가 있다. t_L은 이전의 이벤트 시간을, t_N은 다음 이벤트 시간을 나타내는 시간 변수이다. 각 coordinator는 하위 시뮬레이터들의 t_N값 중 가장 가까운 시간 값을 done-메시지에 실어 올려 보내고, 최상위 root-coordinator는 t_L과 t_N의 시간 차이만큼 시뮬레이션 시간 값을 갱신하고 *-메시지를 내려 보냄으로써 다음 이벤트 스케줄링을 시작한다.

2.2 시뮬레이션과 애니메이션

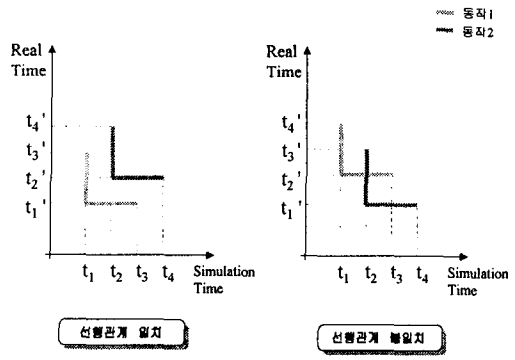
시뮬레이션과 애니메이션은 각각 독립된 분야이지만, 이들이 서로 밀접하게 연관되어 상호상승 효과를 얻을 수 있다. 이들을 결합하는 데는 두 가지 형태가 있는데, 한 형태는 효율적인 애니메이션 제어를 위해 시뮬레이션 이론을 적용하여 움직이는 객체의 동작이 시뮬레이션 모델에 명시된 것에 의해 정해지게 하는 것이다[15]. 다른 형태는 시뮬레이션을 효과적으로 표현하기 위해 애니메이션을 이용하는 방식[1,10,14]으로써 애니메이션은 통계적인 수치 값으로는 찾아내기 힘들거나 불가능한 시뮬레이션의 과정 및 결과를 가시화하여 시뮬레이션을 쉽게 이해하도록 한다[14]. 전자의 형태는 아직 연구적으로 이루어지고 있고, 후자의 형태는 이미 일반화되어 상용화된 시뮬레이션 개발 환경에 적용되고 있다.

최근 10여 년 동안 시뮬레이션 애니메이션 소프트웨어 제품들이 많이 쏟아져 나왔다. 이들 소프트웨어는 시뮬레이터에 독립적인 것도 있고, 시뮬레이션 대상 시스템에 독립적인 것 등 그 형태가 다양하다.

이들을 시뮬레이션과 애니메이션 연결 방식 관점에서 구분[1]하면, 첫째, 시뮬레이션 프로그램의 결과 데이터를 애니메이션 프로세서의 입력으로 하여 애니메이션을 실행하는 형태이다. 애니메이터가 데이터 파일을 갖고 처리하므로 시뮬레이션 소프트웨어와의 독립성을 보장한다. 둘째, 시뮬레이션과 애니메이션이 모델을 통한 상호작용(interaction) 없이 직접 연결되어 있어 하나의 프로그램으로서의 역할을 하는 형태이다. 시뮬레이터와 애니메이터는 각각 하나의 태스크로 동작할 수 있고, 멀티태스킹 환경에서 동작 할 수도 있다. 셋째, 시스템의 변화를 즉시 화면에 출력하기 위해 시뮬레이션 실행 중에 모델 수정을 허용하는 형태이다. 본 논문의 계층적 애니메이션은 시뮬레이션을 효과적으로 보이기 위한 애니메이션이고 애니메이션 프로그램은 시뮬레이션 대상 시스템에 독립적이며 시뮬레이션 실행도중 사용자와의 상호작용을 배제한 형태의 애니메이션 프로그램이다.

하지만, 이러한 분류에 무관하게 모든 시뮬레이션 애니메이션에 필요한 공통적인 핵심 기술이 있다. 우선 시뮬레이션 시간과 애니메이션 시간을 관리하는 일이다. 시뮬레이션에서 시간은 시간변수와 이벤트에 의해 이산적으로 갱신되지만, 애니메이션에서 시간은 화면을 일정 단위 시간 간격으로(연속적으로) 출력하며 애니메이션 시간이 갱신된다. 이들 두 가지 서로 다른 시간 갱신 방법과 무관하게 서로의 시간 값은 일치 즉, 시뮬레이션과 애니메이션의 이벤트 동기화가 필요하다[17]. 이에 덧붙여 애니메이션의 멀티태스킹 처리 및 애니메이션 동작들간 동기화를 해결하기 위한 기술이 필요하다[1]. 멀티태스킹에 의한 애니메이션 처리의 실시간 선행관계는 시뮬레이션에서의 이벤트 선행관계와 일치하도록 보장하지 않으므로, 이에 대한 동기화 처리는 애니메이션 관련 스케줄링 알고리즘으로 처리가 되어야 한다. 그림 <그림 2>는 시뮬레이션 시간과 실시간

사이의 선행관계를 두 경우로 나타낸 것이다. 시뮬레이션에서 동작1은 t1에 시작하여 t3에 종료되고, 동작2는 t2에 시작하여 t4에 종료되며 각 동작에 대한 애니메이션 지시는 t1, t3에 내려진다. 하지만, 실시간에서 이 두 동작은 시뮬레이션 이벤트 순서와 같이 동작1의 애니메이션이 동작2보다 먼저 시작하여 종료(선행관계 일치) 될 수도 있고, 반대로 동작2가 동작1보다 먼저 시작되고 종료(선행관계 불일치) 될 수 있다.



<그림 2> 시뮬레이션 시간과 실시간의 이벤트 선행관계

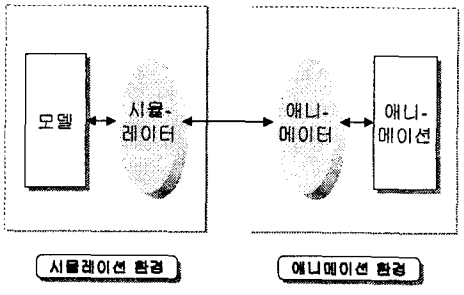
2.3 일반적인 애니메이션의 계층성

계층성이란 용어는 많은 분야에서 사용되고 있지만, 그 의미는 분야별로 사뭇 다르다. 일반적인 애니메이션에서의 계층성의 의미와 본 논문에서 제안하는 시뮬레이션을 위한 애니메이션의 계층성과는 어떤 차이가 있는지 살펴보면, 우선 일반적인 애니메이션에서 계층성은 두 가지 의미로 해석 될 수 있다. 하나는 그래픽적 객체를 보는 위치에 따른 사실적인 표현을 하기 위해 여러 단계로의 복잡도(level of detail)로 모델링하는 것을 말한다. 즉, 동일한 애니메이션 대상에 대해 이를 표현하는 그래픽적인 정도를 조정하게 함으로써 표현할 대상의 출력 용도에 따라 복잡도를 다르게 출력시키는 것이다. 다른 한 측면에서의 계층성은 여러 그래픽 객체를 위치한 좌표를 기준으로 한 우선순위에 따라 겹쳐지는 영역을 앞/뒤로 오게 하여 숨기거나 보이게 하는 것을 의미한다[16]. 그러나, 본 연구에서 계층적 애니메이션이라 함은 그래픽적 상세 정도나 출력 이미지의 우선순위가 아니라, 애니메이션 대상 시스템의 내부를 들여다보는 관점에 따라 특정 계층 또는 구성요소(tree형태의 시스템 구조에서)를 선택하여 애니메이션 함으로써 객체의 현재 상태를 표현한다. 단, 애니메이션 정보는 모델에 명시되어 시뮬레이션에 의해 스케줄링 된 동적 결과에 의존한다.

지금까지의 내용을 토대로 이후 사용될 용어의 의미를 간단히 소개하겠다. *추상 시뮬레이터*는 모델에 명시된 이벤트 스케줄링을 하는 프로세서들이 연결되어 시뮬레이션을 위한 구조화된 프로세서 역할을 하는 것을 의미하고, *추상 애니메이터*는 모델별로 존재하는 애니메이터들이 연결되어 애니메이션 엔진 역할을 하는 구조화된 애니메이션 프로세서를 의미하며, *시뮬레이션 애니메이션 환경*은 시뮬레이션부터 이를 반영하는 애니메이션 동작까지의 시뮬레이션 개발에 필요한 모든 구성요소가 합쳐진 것을 의미한다.

3. 계층적 애니메이션 환경 설계

계층적 애니메이션을 하기 위한 전체적인 필요 구성 요소들을 살펴본다. 특히, 핵심 구성 요소인 계층적 애니메이터의 필요성을 3.3절에서 자세히 기술한다.



<그림 3> 시뮬레이션과 애니메이션 관계

3.1 개념적인 구성 모듈

시뮬레이션을 위한 계층적 애니메이션에서는 <그림 3>과 같이 크게 네 개의 요소로 구성된다. 시스템의 동적인 특징과 이를 애니메이션으로 출력하기 위한 명세인 모델, 시뮬레이션의 이벤트 스케줄링을 담당하는 시뮬레이터, 애니메이션 동작들에 대한 스케줄링을 하는 애니메이터, 애니메이션을 화면에 출력시키는 환경인 애니메이션이 전체 시뮬레이션 구성 요소가 된다.

애니메이션으로 표현하는 시뮬레이션 개발 환경에서는 모델링 할 때 시스템의 이벤트 스케줄을 위한 정보와 각 이벤트에 의한 상태변화를 애니메이션하기 위한 정보를 정의해야 한다. 정의된 모델마다 시뮬레이터와 애니메이터를 갖게 함으로써 모델구조의 계층성을 애니메이션에 그대로 반영할 수 있다.

3.2 계층적 모델

시스템을 계층적으로 표현하기 위한 모델은 2.1에서 설명한 DEVS 형식론에 기반하여 정의하는 것을 기본으로 하고, DEVS에서의 기본 모델과 결합모델 각각에 다음의 애니메이션 관련 정보들을 추가하여 모델의 상태가 변할 때 이미지의 출력 및 동작에 대한 명세를 함으로써 계층적 애니메이션 출력이 있는 시뮬레이션 모델을 정의한다.

- ImgObjs : 움직임 없는 이미지 객체리스트, 모델을 표현하는 배경이미지
- AnimObjs: 애니메이션 객체리스트, 모델의 동적 상태 출력
- DispWnd : ImgObjs, AnimObjs가 출력될 윈도우

기본모델은 모델의 상태를 그래픽으로 출력하기 위해 상태변수의 일부에 ImgObjs, AnimObjs, DispWnd의 값들을 추가한다. 결합모델은 그래픽 정보를 나타내기 위한 정보를 모델 의 한 구성요소로 추가하여 모델을 그래픽으로 출력하게 하는데, 결합모델의 특성상 AnimObjs를 제외한 ImgObjs와 DispWnd 정보만이 명시된다. 기본모델에서 위의 AnimObjs는 δ_{int}, δ_{ext}를 통해 모델의 동적 특성(dynamics)에 적절하게 구체적인 움직임을 지시 받는다.

3.3 계층적 시뮬레이터

시스템을 계층적으로 표현하고 이에 상응하는 시뮬레이션 엔진이 있어야 하며 이 또한 모델의 계층성을 그대로 반영하기 위해서는 계층적인 형태로 만들어져야 한다.

2.1절에서 시뮬레이션 환경 중 DEVS형식론을 충실히 반영한 것으로 DEVS-scheme을 예로 들었으며, 계층적 애니메이션 설계를 위한 시뮬레이션 개발 환경으로 이 DEVS-scheme을 가정하였다. 그러나, 실제 구현에서는 애니메이션 처리를 위해 C++을 사용하므로 시뮬레이션과의 연결을 용이하게 하기 위해 DEVS 형식론을 C++로 작성한 DEVS-C++을 이용한다. DEVS-C++은 애니메이터와의 연결을 위해 일부를 수정 및 추가 해야하는데, 이는 4장에서 계층적 애니메이터의 자세한 설계와 함께 설명한다.

3.4 계층적 애니메이터

일반적인 애니메이션이 각 모델에 있는 애니메이션 객체들을 일괄적으로 관리하면서 화면에 출력하는 것과 달리, 계층적 애니메이션은 시뮬레이션 모델마다 각 애니메이션 객체들을 갖고 이들을 제어하는 프로세서인 애니메이터를 갖는다. 애니메이터는 기본 모델에 해당하는 것과 결합모델에 해당하는 것의 형태가 각기 다르고, 각 모델에 해당하는 애니메이터가 생성된다. 개별적인 애니메이터 생성은 애니메이터 구조가 자연스럽게 모델의 계층구조를 갖도록 하는데, 이렇게 형성된 애니메이터를 계층적 애니메이터라 한다. 이러한 애니메이터의 구조가 모델과 같은 계층성을 갖는 것은 다음과 같은 이유로 그 필요성이 요구된다.

첫째, 시뮬레이션과 애니메이션간의 동기화를 용이하게 한다. 이산적으로 시간이 경과되는 시

뮬레이터는 연관된 모델의 상태변화에 따라 애니메이션 지시를 내리고, 시뮬레이터가 다음 이벤트를 처리하기 전에 애니메이터로부터 이전 이벤트에 대한 애니메이션 처리가 끝났음을 알리는 메시지를 받게 함으로써 시뮬레이션과 애니메이션 상호간 이벤트의 동기화를 이룰 수 있다. 이러한 동기화의 대상이 모델과 같은 계층성을 갖는 시뮬레이터이므로 애니메이터 또한 그 계층성을 따르면 동기화를 쉽게 할 수 있다. 즉, 각 모델은 직접 연관된 상위 및 하위 모델만 고려하여 애니메이션 관련 스케줄링을 하면 된다.

둘째, 시스템의 계층별 명세작성을 용이하게 한다. 모델에 애니메이션 정보를 함께 명세화해도 무리가 없도록 하기 위해서, 이를 뒷받침 할 수 있는 애니메이션 프로세서(애니메이터)가 필요하다. 따라서 모델구조가 계층적이므로 애니메이터가 계층적인 구조를 가져야 한다.

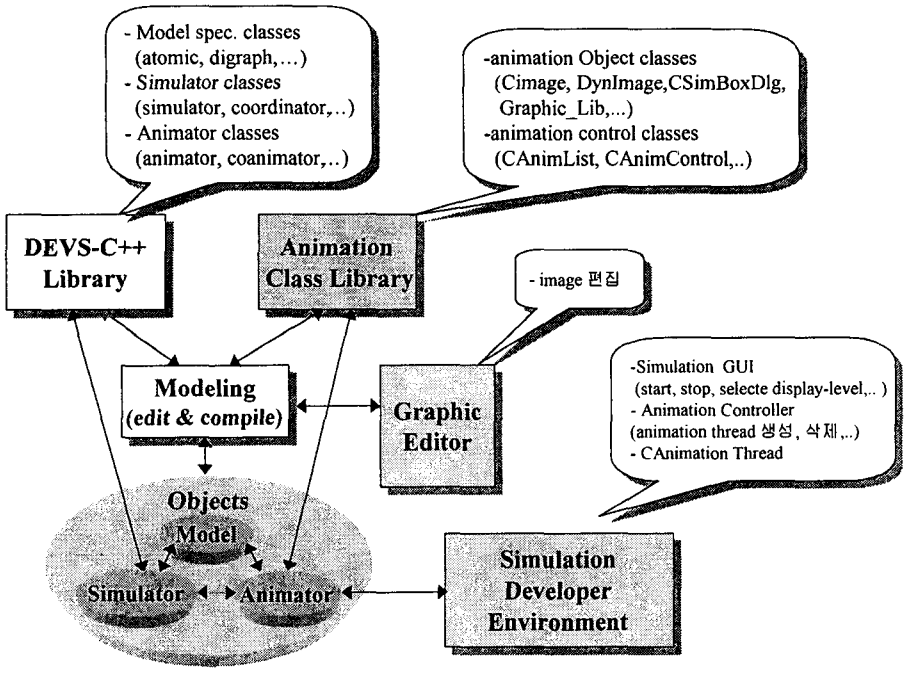
셋째, 애니메이션 모델들의 재사용이 용이하다. DEVS형식론에 의한 시뮬레이션은 모듈성이 좋게 모델링을 할 수 있으므로, 만들어 놓은 모델들의 재사용 및 재구성을 쉽게 할 수 있다. 애니메이션을 요구하는 모델들도 애니메이션이 없는 기존의 DEVS 모델처럼 모델의 재사용이 가능하려면, 시뮬레이터가 모델마다 독립적으로 생성되듯 애니메이터도 모델마다 독립적으로 생성되어야 한다. 이렇게 하기 위해서는 계층적인 모델 구조를 그대로 따르는 계층적 애니메이터의 필요성이 요구된다.

이 외에도 DEVS 시뮬레이션 모델링의 계층적이고 객체 지향적인 설계가 지닌 장점들을 애니메이션 모델링에서도 반영되도록 할 수 있다.

3.5 애니메이션 출력 환경

계층적 애니메이션 개발 환경을 포함했을 때의 전체적인 시뮬레이션 개발 환경은 <그림 4>와 같다.

모델링을 할 때 시뮬레이션과 애니메이션에 관



<그림 4> 시뮬레이션 애니메이션 개발 환경 구성도

련된 것들은 각각 DEVS-C++ 라이브러리, Animation Class 라이브러리를 이용하여 작성하고, 애니메이션 관련 모델링을 할 때 필요한 그래픽 관련 파일들은 Graphic Editor로 만든다. 이렇게 하여 생성된 시뮬레이션 모델이 컴파일되면, 사용자가 작성한 모델의 정보를 유지하는 Model, 이벤트 스케줄링을 하는 프로세서인 Simulator, 이벤트 스케줄링을 가시화하기 위한 프로세서인 Animator 들이 생성된다. 실제로 시뮬레이션 진행은 이러한 프로세서들 간의 상호작용을 통해 이루어진다.

4. 계층적 애니메이터 설계

계층적 애니메이터의 핵심 기능은 이벤트 스케줄링과 이를 표현하는 애니메이션 사이에 동기화를 시켜주는 일이다. 이산 사건 모델에서 시뮬레이션 시간은 시간 변수 값을 이산적으로 증가시킴으로써 이루어지는 반면, 애니메이션에서 시간은 연속적인 그래픽 객체의 출력으로 이루어진다. 즉, 모델에서 스케줄링 후 갱신되는 시간만큼 애니메이션이 진행되며 실시간이 흐른다. 이러한 서로 다른 시간 갱신을 고려 할 때 애니메이터가 시뮬레이터와의 상호작용(interaction)을 위해서는 이들간의 시간 동기화가 무엇보다 중요하게 처리되어야 할 사항이다. 또, 모델의 재구성을 반영하기 위해 추상 시뮬레이터 구조가 모델 구조를 반영하여 생성되듯 추상 애니메이터 또한 모델의 구조를 반영하여 형성되도록 한다.

이러한 것들을 가능하게 하기 위해 각 애니메이터에 필요한 정보를 <표 1>에서와 같이 유지시키고, 이들의 정보는 애니메이터간 메시지 교환을 통해 설정 및 변경이 이루어지며 메시지 형태는 <표 2>와 같이 정의하였다. 또, 그림이나 표에서 사용할 시뮬레이션 및 애니메이션 관련 프로세서들을 간략히 표시하기 위한 약어를 <표 3>에서와 같이 정의한다.

애니메이터는 시뮬레이터와의 동기화를 위해서 이전 이벤트의 시간 변수(t_{LA})와 다음 이벤트의 시간 변수(t_{NA})를 갖는다. t_{LA}, t_{NA} 값은 시뮬레이터에서 모델에 의해 명시된 내용대로 이벤트 스케줄링 할 때의 시뮬레이션 시간 값과 같이 유지되는데, 이들이 갱신될 값들은 시뮬레이터로부터 또는 다른 애니메이터로부터의 메시지를 통해 전달받는다. root-co-animator와 co-animator의 t_{LA}, t_{NA}는 (imminent(s), t), ($pause_A$, t), (done, t_{NA}, source, t) 메시지들을 통해 전달된다. 단, Animator는 해당 DEVS모델의 t_L, t_N을 직접 참조한다.

<표 1> 동기화를 위한 정보

Sync. Information	의 미
t_{LA}	마지막 애니메이션을 끝낸 시뮬레이션 시각
t_{NA}	다음 애니메이션의 시뮬레이션 시각
L	이벤트 스케줄링이 있는 children. (*, x-메시지 받는 모델)
AL	애니메이션 스케줄링이 있는 children. (done보낸 것 중 애니메이션 동작이 있는 모델)
$done_A$-list	애니메이션이 끝난 후 동기화를 위해 $done_A$-메시지를전달하지 않은 시간정보 리스트.
to-be-$done_A$-list	$done_A$-메시지를 받아야 하는 시간 값 리스트

<표 2> 계층적애니메이터의 메시지 타입

Message Type	의 미	Sender -> Receiver
Imminent	이벤트 스케줄링 시작을 알림	S->A
done	시뮬레이터에서 이벤트 처리가 끝났음을 보고	S->A
$*_A$	애니메이션을 시작 지시	A->A
$done_A$	애니메이션이 끝났음을 보고.	A->A A->S
$pause_A$	진행 중이던 애니메이션 중단 지시	A->A
ask-$done_A$	이미 완료된 애니메이션이 있는지 확인	A->A

<표 3> 시뮬레이터 및 애니메이터 관련 약어

약 어	프로세서	설 명
R	Root-co-ordinator	시뮬레이션의 시간제어 담당
C	Co-ordinator	결합 모델에 대응되는 시뮬레이터
S	Simulator	기본 모델에 대응되는 시뮬레이터 또는 R,C,S 모두를 지칭.
RA	Root-co-Animator	Root-coordinator와 대응되는 애니메이터
CA	Co-Animator	Coordinator와 대응되는 애니메이터
A	Animator	Simulator와 대응되는 애니메이터 또는 RA,CA, A모두를 지칭

애니메이터는 이벤트 스케줄이 일어난 것과 관련된 애니메이션 스케줄을 하기 위해 이벤트 스케줄을 행한 모델들을 식별할 수 있는 정보를 유지하는데, 변수 L과 AL이 그것이다. L은 root-co-animator, co-animator에서 하위 애니메이터들에 대응되는 시뮬레이터가 *-메시지를 받았는지(내부 상태변이 발생)에 대한 정보를, AL은 done-메시지를 보고했는지(내부 및 외부 상태변이 발생)에 대한 정보를 나타낸다.

<표 4> 메시지 전달 형태

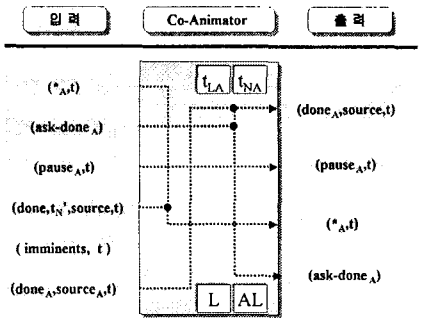
표현양식	의미
	FROM/TO 상위 애니메이터
	FROM/TO 대응되는 시뮬레이터
	FROM/TO 하위 애니메이터
(.....)	메시지가 전달하는 정보
------▶	애니메이터에서 처리되는 메시지 변환경로

그 외 애니메이션에서 이벤트 선행관계가 시뮬레이션에서의 이벤트 선행관계와 불일치 한 경우를 처리하기 위한 정보 및 메시지가 필요하다. 2장의 <그림 2>의 경우처럼 시뮬레이션 이벤트 순서는 동작1이 동작2보다 먼저 시작하고 종료하는 반면, 애니메이션 처리를 하는 실제 시간에는 동작2가 먼저 시작하고 종료될 수 있다.

애니메이션의 이벤트 선행관계 문제를 해결하기 위해 애니메이터는 $done_A$의 시간 리스트($done_A$-list)와 $done_A$를 받아야 할 시간 리스트(to-be-$done_A$-list)를 관리하는데, 이는 애니메이션 스케줄이 이루어질 때마다 갱신된다. 또, 이들 값을 애니메이션 스케줄에 반영하기 위해 as-$done_A$-메시지를 추가로 정의한다. 상위 애니메이터에게 $done_A$-메시지를 전달하는 시점은 반드시 하위 애니메이터로부터 애니메이션 지시에 대한 t_{NA}에 해당하는 모든 $done_A$-메시지를 받았을 때이고, 기다리는 $done_A$보다 이후에 끝나는 애니메이션이 $done_A$-메시지를 보냈을 때는 $done_A$-list에 저장한다. ask-$done_A$-메시지는 root-co-animator가 $*_A$-메시지를 보내고 t_{NA}에 해당하는 $done_A$를 기다리기 전에 이미 완료됐는지를 확인하기 위해 하위 애니메이터에게 $done_A$-list를 검색하라는 지시를 내리고, 검색된 것이 있으면 to-be-$done_A$-list에서 삭제함으로써 시뮬레이션에서의 이벤트 선행관계가 애니메이션에 반영되도록 한다.

이후는 추상 애니메이터를 이루는 root-co-animator, co-animator, animator에 대한 구체적인 스케줄링 방법 및 시뮬레이터의 수정내용을 pseudo 코드와 함께 설명하고, 그림에서 표현 할 메시지 입출력의 형태는 <표 4>와 같다.
태는 다음과 같다.

4.1 Root-Co-Animator



<그림 5> Co-Animator의 메시지 입출력

1) imminent-메시지 처리

root-coordinator로부터 (imminent(s), t)메시지를 받는다.

if $t = t_{NA}$,

$t_{LA} = t$

$L = L \cup imminent(s)$

t는 root-coordinator가 이벤트 스케줄링을 한 시간이며, t_{NA}은 최근에 애니메이션이 끝난 시간을 의미한다. 이 두 값이 같을 때(동기화 확인) 이전 애니메이션 시간을 현재 스케줄링 시간으로 갱신하고, 이벤트 스케줄링이 행해진 시뮬레이터에 대응되는 하위 애니메이터(source)를 L에 추가한다.

2) done-메시지 처리

root-coordinator로부터 (done, t_{NA}' , source, t) 메시지를 받는다.

$t_{LA} = t$
$t_{NA} = t_{NA}'$
if L = Ø ,
Issue ($*_A$, t) downward
if imminent's type is co-animator ,
Issue (ask-done$_A$) downward

시뮬레이터에서 시간 t에 해당하는 모든 이벤트 스케줄링이 끝났으면(L = Ø) 하위 애니메이터에게 **$*_A$-메시지를 전달하고**, 하위 애니메이터가 co-animator이면 **ask-done$_A$-메시지를 전달한다**.

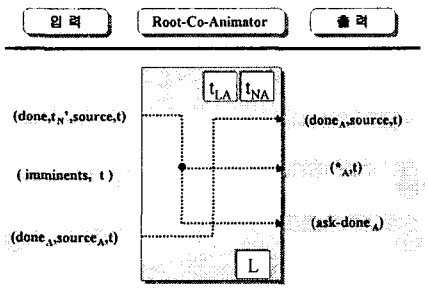
3) done$_A$-메시지 처리

하위 애니메이터로부터 (done$_A$, source, t)메시지를 받는다.

if t = t_{NA} ,
Issue(done$_A$, NULL, t_{NA}) to root-coordinator

하위 애니메이터의 done$_A$ 시간 t가 t_{NA}와 일치하는가를 확인 한 후, root-coordinator에게 **done$_A$-메시지를 전달한다**

4.2 Co-Animator



<그림 6> Co-Animator의 메시지 입출력

Co-ordinator, Root-co-Animator, 하위 애니메이터들과의 연결을 위해 6가지 유형의 메시지 (imminent, done, $*_A$, done$_A$, pause$_A$, ask-done$_A$) 처리가 있다. 이 중 세 개의 메시지(imminent, done, done$_A$)가 시뮬레이션의 이벤트 시간과 애니메이션 간 동기화를 수행한다. Imminent, pause$_A$-메시지는 t_{LA}값을 갱신하며, done-메시지는 t_{NA} 값을 갱신한다. 나머지 세 개(done$_A$, $*_A$, ask-done$_A$)는 애니메이션의 시작과 끝을 알리고 애니메이션 간 동기화를 수행한다.

1) imminent-메시지 처리

coordinator로부터 (imminent(s), t)메시지를 받는다.

if t = t_{NA} ,
t_{LA} = t
L = L ∪ imminent(s)

coordinator가 이벤트 스케줄링을 한 시간 t와 최근에 끝난 애니메이션 시간 t_{NA}으로 동기화가 이루어졌음을 확인하고, 이전 애니메이션 시간을 현재 스케줄링 시간으로 갱신한 후 L에 s(source)를 추가한다.

2) done-메시지 처리

coordinator로부터 (done, t_{NA}' , source, t)메시지를 받는다.

if $t_{NA}' = 0$,
L = L ∪ {source}
else if source∈L ,
L = L - {source}
AL = AL∪{source}
else if source∉L and t=t_{LA} ,
AL = AL∪{source}
else if source∉L and $t_{LA} < t \le t_{NA}$,
AL = AL∪{source}
A-flag = false
Issue (pause$_A$, t) downward
if source is on animation
delete source's t_{NA} to-be-done$_A$-list
insert t_{NA}' in to-be-done$_A$-list
When $t_{NA}' < t_{NA}$ then
$t_{NA} = t_{NA}'$

시뮬레이션 초기에 source의 t_{NA}을 의미하는 t_{NA}' 이 0이고, done-메시지가 내부상태변이의 결과에 의한 것 일 때 source는 L에 들어있다 (source∈L). source∉L는 외부입력에 의한 이벤

트 스케줄링 의미하고 $t=t_{LA}$는 하위 애니메이터들 중 현재 진행중인 애니메이션이 없음을 의미하는데, 내부상태변이가 같은 시간에 발생한 경우에 해당한다. 외부 입력에 의한 이벤트 스케줄링이 있었고 $t_{LA}<t\leq t_{NA}$이라면, 현재 애니메이션이 진행중인 하위 애니메이터가 있음을 의미한다. 이때는 새로운 이벤트에 의한 애니메이션을 반영하기 위해 진행중인 애니메이션을 중지하도록 하위 애니메이터에게 $pause_A$-메시지를 전달하여 다음 $*_A$-메시지 올 때 외부입력을 반영한 애니메이션을 수행하도록 하고, source의 이전 t_{NA}값을 to-be-$done_A$-list에서 제거하고 새로운 t_{NA}' 값을 to-be-$done_A$-list에 넣는다. 마지막으로 외부상태변이에 의한 t_{NA}값을 갱신한다.

3) $*_A$-메시지 처리

하위 애니메이터로부터 ($*_A$, t)메시지를 받는다.

```
if  L = Ø  and  t = tLA,
        A-flag = true
           Issue *A to all AL downward
           add tNA of  AL children to to-be-doneA-list
           Reset AL to Ø
```

이벤트 스케줄과의 동기화를 확인 한 후 애니메이션 처리가 있는 하위 애니메이터에게 $*_A$-메시지를 전달하고, 애니메이션 지시를 내리면서 $done_A$-메시지 받아야 할 시간 값을 to-be-$done_A$-list에 추가한다.

4) $done_A$-메시지 처리

하위 애니메이터로부터 ($done_A$, source, t)메시지를 받는다.

```
if   t = tNA ,
   delete tNA from to-be-doneA-list
   If  there is no more tNA in to-be-doneA -list
       Issue (doneA, tNA) upward
    Else
         Return;
else if  t > tNA ,
   add t to done-tNA-list
```

하위 애니메이터의 $done_A$ 시간 t가 t_{NA}와 일치하면 to-be-$done_A$-list에서 해당 시간을 삭제하고, t_{NA}에 종료되어야 할 애니메이션 동작이 모두 완료됐으면 상위 애니메이터에게 $done_A$-메시지를 전달한다.

하위 애니메이터에서 애니메이션 종료시간 t가 다른 진행 중인 애니메이션의 종료시간 t_{NA} 이후의 시간이라면($t > t_{NA}$), t_{NA}가 t의 값과 같을 때까지 $done_A$-list에서 기다린다.

5) $pause_A$-메시지 처리

상위 애니메이터로부터 ($pause_A$, t)메시지를 받는다.

```
if  tLA<t≤tNA  and  A-flag ,
        tLA= t;
        pause animation
     A-flag = false;
     Issue (pauseA, t) downward
     delete tNA from to-be-doneA-list
```

외부상태변이가 일어났는데 현재 애니메이션 중이면 진행중인 애니메이션을 멈추게 하기 위해 하위 애니메이터에게 $pause_A$-메시지를 전달하고, t_{NA}를 to-be-$done_A$-list에서 지워 t_{NA}에 끝날 애니메이션에 대한 $done_A$를 기다리지 않도록 한다.

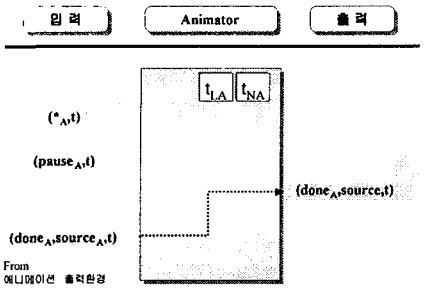
6) ask-$done_A$-메시지 처리

상위 애니메이터로부터 (ask-$done_A$)메시지를 받는다.

```
if  doneA-list≠Ø  ,
          delete tNA from doneA-list
          delete tNA from to-be-doneA-list
if  to-be-doneA-list = Ø ,
          Issue (doneA, null, tNA) to this
else
          Issue (ask-doneA) downward
```

이미 애니메이션이 종료된 것이 있으면 $done_A$-list에서 t_{NA}에 해당하는 것을 모두 삭제하고, $done_A$-list에 있던 t_{NA}의 개수만큼 to-be-$done_A$-list에서의 t_{NA}를 삭제한다. 그리고, 더 이상 기다릴 $done_A$-메시지 없으면 자기 자신에게 $done_A$-메시지를 전달하며, 그밖에는 하위 애니메이터에게 ask-$done_A$-메시지 전달한다.

4.3 Animator



<그림 7> Aniamtor의 메시지 입출력

상위 애니메이터와의 연결을 위해 2가지 유형의 메시지($*_A$, pause$_A$)를 처리하고, 애니메이션 출력 환경으로부터 오는 done$_A$메시지를 처리한다. $*_A$는 애니메이션 시작을 지시하기 위한 메시지이고, pause$_A$는 외부상태변이를 반영하기 위해 진행중인 애니메이션을 멈추게 한다. t_{LA}, t_{NA}는 각각 모델의 상태변이 함수가 실행된 후 값이 갱신되고, t_{NA}는 pause$_A$-메시지에 의해 한 번 더 갱신된다.

1) $*_A$-메시지 처리

하위 애니메이터로부터 ($*_A$, t)메시지를 받는다.

```
if   this is waiting for done_A-msg,
     Return;
if   t = t_LA ,
        Sigma = t_NA - t_LA
        Start Animation for Sigma
```

$*_A$-메시지는 done$_A$-메시지 받은 후 받아야 한다는 규칙이 지켜졌는지 확인한 후, 시뮬레이터에서 상태변이함수 실행 후에 갱신된 t_{LA} 값이 만족하면 애니메이션 출력 환경에 다음 이벤트까지의 사간 동안의 애니메이션을 지시한다.

2) pause$_A$-메시지 처리

상위 애니메이터로부터 (pause$_A$, t)메시지를 받는다.

```
if  t_LA≤t≤t_NA ,
        if  A-flag  then
          Pause-animation
          A-flag = false;
t_LA = t;
```

외부 상태변이를 반영하기 위해 현재 진행중인 애니메이션을 멈추고, t_{LA} 값을 갱신한다.

3) done$_A$-메시지 처리

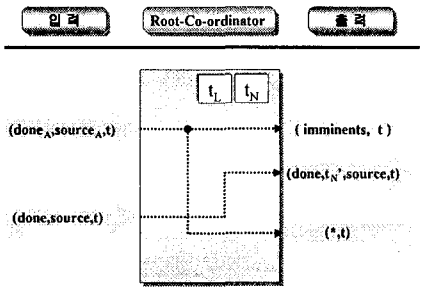
애니메이션 출력 환경으로부터 (done$_A$, source, t)메시지를 받는다.

```
if  t = t_NA
        A-flag = false
        Issue (done_A, t) to parent processor
else
        Return;
```

시간 동기화를 확인한 후 상위 애니메이터에게 done$_A$-메시지를 전달한다.

4.4 시뮬레이터 수정

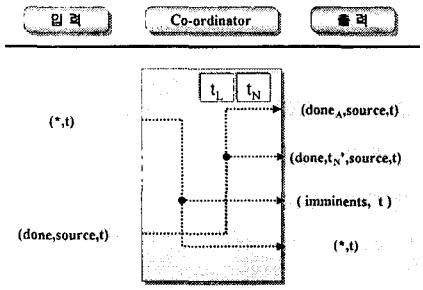
1) root-coordinator



<그림 8> root-co-ordinator의 입출력

시간의 갱신은 항상 갱신될 시간만큼 애니메이션이 끝났다는 done$_A$-메시지를 받고 행해지며 이벤트 스케줄링을 하는 *-메시지를 보낼 때 root-co-animator에게 imminent-메시지를 보낸다. 또 하위 프로세서에게서 이벤트 처리가 끝났음을 done-메세지를 통해 받으면 root-co-animator에게 done-메시지를 보낸다.

2) Coordinator



<그림 9> co-ordinator의 메시지 입출력

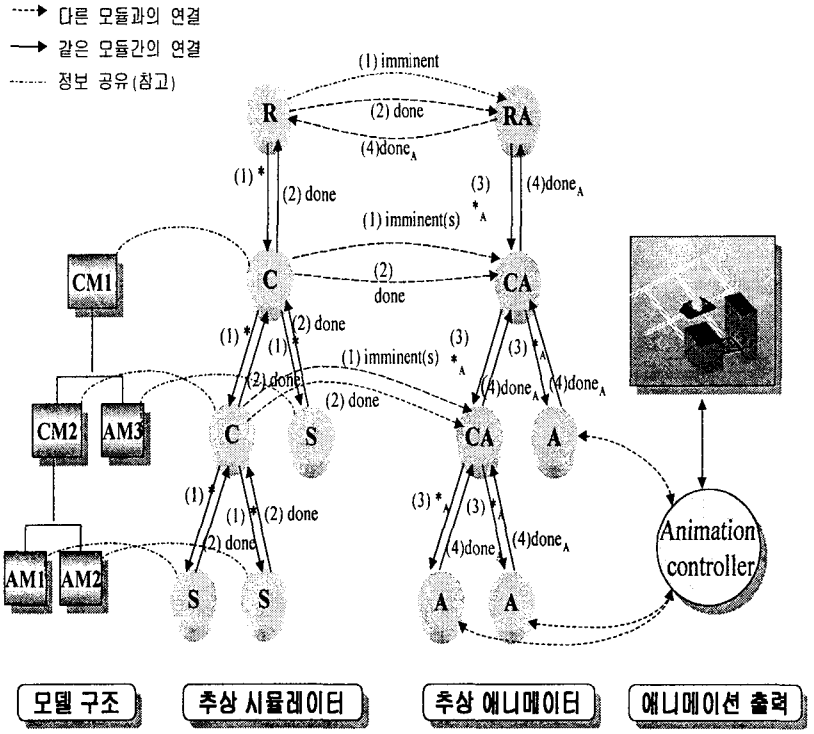
이벤트 스케줄링을 지시하는 *-메시지를 하위 프로세서에게 보낼 때 co-animator에게 imminent -메시지를 보낸다. 또 하위 프로세서로부터 이벤트 처리가 끝났음을 done-메세지를 통해 받아 상위 프로세서에게 전달 할 때 co-animator에게 done-메시지를 보낸다.

3) simulator

상태변이 함수가 수행 될 때마다 변경되는 값을 애니메이터의 t_{LA}, t_{NA}값으로 변경시킨다.

4.5 시뮬레이션 동작 사이클

<그림 10>은 3계층구조를 갖는 시스템의 경우를 예로 각 시뮬레이션 구성 모듈간의 연관성을 나타낸 것이다. 모델 구조에는 이벤트 스케줄링 및 애니메이션 관련 정보, 추상 시뮬레이터는 이벤트 스케줄링을 제어하기 위한 정보, 추상 애니메이터에는 이벤트 스케줄링을 애니메이션에 반영하기 위한 제어 정보, 애니메이션에는 화면



<그림 10> 추상 시뮬레이터와 추상 애니메이터의 메시지 전달

출력에 관한 프로그램이 있다. 이 모든 모듈이 연결되어 시뮬레이션이 수행되는 수행 순서를 한 사이클을 기준으로 살펴보면 크게 4단계로 나눌 수 있고, 그림의 각 메시지 앞의 번호는 각 단계를 표시한 것이다.

첫번째 단계는 R로부터 S까지의 *-메시지 전달이다. 각 시뮬레이터는 모델에 명시된 정보를 이용해 *-메시지를 처리한다. R, C는 *-메시지를 하위 프로세서에게 내려보내는데, 해당 애니메이터에게 imminent-메시지를 보내어 해당 RA, CA의 L을 갱신하게 한다.

두 번째 단계는 S에서 R까지의 done-메시지 전달이다. 해당 시간의 이벤트 처리가 끝난 시뮬레이터는 상위 시뮬레이터에게 done-메시지를 보고하는데, 이 메시지를 받은 시뮬레이터는 해당 애니메이터에게 done-메시지를 보낸다. 이를 전달 받은 RA, CA는 AL을 갱신한다.

RA가 done-메시지를 받았을 때는 이벤트-스케줄링은 끝난 상태이며, 이 때부터 RA가 하위 애니메이터에게 $*_A$-메시지를 보냄으로써 애니메이션 스케줄을 시작한다.

세 번째 단계는 RA에서 A까지의 $*_A$-메시지 전달이다. RA, C는 AL의 모든 하위 애니메이터에게 $*_A$-메시지를 전달한다. 이때 $*_A$메시지를 받은 애니메이터는 출력 레벨이 자신과 일치하는지를 확인하여 $*_A$-메시지를 계속 전달할 것인지를 판단한다. 최하위 애니메이터(A)가 $*_A$-메시지를 받았을 때는 최종적으로 애니메이션 환경에 $*_A$-메시지를 전달한다.

$*_A$-메시지를 받은 애니메이션 환경에서는 이에 대한 동작을 하고, 다 끝나면 해당 메시지를 보낸 애니메이터에게 $done_A$-메시지를 전달한다.

네 번째 단계는 A에서 RA까지의 $done_A$-메시지 전달이다. $done_A$-메시지를 받은 RA는 이 메시지를 R에게 전달한다.

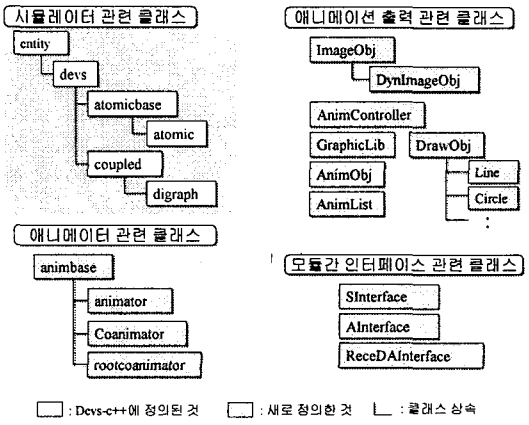
R은 $done_A$-메시지를 받음으로써 이벤트 스케줄링에 대한 애니메이션 처리가 모두 끝났음을 보고 받고, 다음 이벤트 처리를 위해 시뮬레이션 사이클 첫번째 단계부터 다시 진행한다.

5. 구현 및 검증

3.4절의 전체 설계에서 밝힌 것과 같이 시뮬레이터로는 DEVS-C++을 사용하고, 애니메이션을 위한 그래픽 편집기 및 라이브러리 함수는 자체 개발한 것을 사용하며 개발 환경으로는 그래픽 사용자 인터페이스 개발에 용이한 VisualC++를 이용하였다. 이런 환경하에서 간단한 제조공정시스템을 예로 구현을 검증한다.

5.1 구현

1) 모듈별 주요 클래스



<그림 11> 모듈별 주요 클래스

<그림 11>은 계층적 애니메이션을 하기 위한 각 모듈의 주요 클래스를 나타낸 것이다. 시뮬레이터 관련 클래스는 DEVS-C++에서의 기본모델 타입인 atomic 클래스와 결합모델인 digraph 클래스까지의 클래스를 나타낸 것이다. DEVS-C++에서는 모델클래스 내에 Simulator, Coordinator, Root-Coordinator 역할을 담당하는 부분이 멤버함수로 정의되어 있다. atomic 클래스 내에 Simulator에 해당하는 함수가 있고 digraph 클래스 내에 Coordinator에 해당하는 함수가 있으며, atomic과 digraph 모두 최상위모델이 될 수 있으므로 Root-Coordinator에 해당하는 함수가 있다.

애니메이터 관련 클래스는 애니메이터 타입들의 공통적인 내용으로 정의된 animbase 클래스가 있고, 이를 상속하여 animator, coanimator, rootcoanimator 클래스를 정의한다.

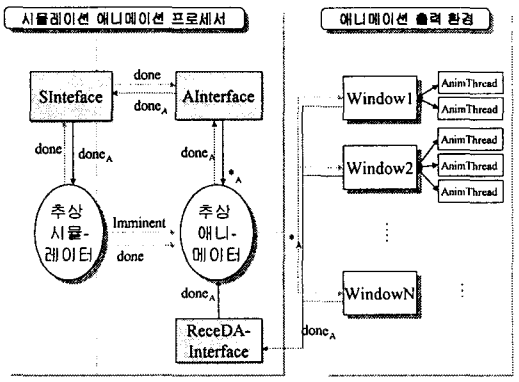
애니메이션 출력과 관련한 클래스에는 우선 화면에 그려질 이미지를 정의하기 위한 ImageObj 클래스가 있고, 애니메이션 동작을 하기 위해 ImageObj 클래스를 상속하여 정의한 DynImage Obj 클래스가 있다. 각 이미지는 여러개의 도형이 조합되어 생성되는데, 각 도형을 정의하기 위해 DrawObj 클래스 및 이를 상속한 Line, Rectangle, Circle,... 등의 클래스가 있다. 각 이미지는 이미지 편집기에 의해 그려지고, 이들을 애니메이션 출력하기 위한 ImageObj 및 DynImage Obj로 인식하기 위한 GraphicLib 클래스가 있다. 이렇게 생성된 DynImagObj를 애니메이션 정보 단위로 관리하기 위해 AnimObj 클래스를 정의하고, 애니메이션 정보들을 관리하기 위한 AnimList 클래스가 있으며, DynImageObj를 동작시키기 위한 AnimController 클래스가 있다. 모듈간 인터페이스 클래스는 다음 3)절과 같다.

2) 애니메이션 병렬처리

전체 시뮬레이션 모델의 애니메이션간 병렬 처리는 멀티 쓰레드 기법을 이용한다. 각 모델별 애니메이션 동작이 해당 시뮬레이터의 이벤트 스케줄링을 반영한 애니메이터에 의해 시작 및 종료되고, 동작이 시작될 때마다 독립된 애니메이션 쓰레드를 생성한다.

3) 모듈간 인터페이스

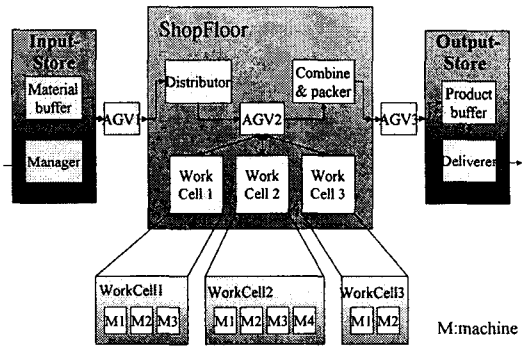
애니메이터 모듈, 시뮬레이터 모듈, 애니메이션 출력 환경 모듈을 연결하는데, 시뮬레이션과 애니메이션의 동기화 및 애니메이션 간 동기화를 하기위해 인터페이스 모듈을 [그림 5-2]에서와 같이 정의한다. 추상 시뮬레이터와의 인터페이스를 위한 SInterface, 추상 애니메이터와의 인터페이스를 위한 AInterface, 애니메이션 출력 환경으로부터 $done_A$-메시지를 받아 추상 애니메이터에게 전달하는 인터페이스를 위한 ReceDA-Interface가 있다. 추상 애니메이터에서 애니메이션 지시를 내리는 $*_A$-메시지는 인터페이스를 거치지 않고 직접 지시하는데, $*_A$-메시지는 이미 스케줄링이 다 끝난 상태이고 애니메이션 동작에서 다른 스케줄링이 없기 때문이다.



<그림 12> 모듈간 인터페이스

5.2 테스트

1) 시스템 구성도



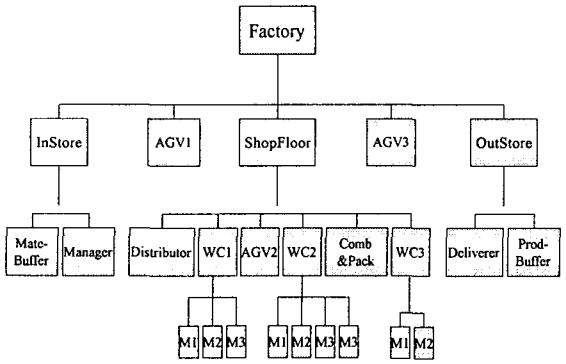
<그림 13> 제조 공정 시스템 구성도

<그림 13>은 단순화시킨 제조 공정 시스템의 구성도로서 3가지 부품을 생산하고 조립하여 하나의 제품을 만드는 시스템을 계층성을 표현할 수 있도록 임의로 구성한 것이다. 크게 자재 반

입, 생산 공장, 제품 납입부분과 이들 간에 물건을 운반하는 두 대의 AGV로 구성되어 있다. 자재 반입은 외부로부터 자재를 들여오는 부분과 자재창고로 구성되고, 제품 납입은 제품창고와 외부로 출하하는 부분으로 구성된다. 생산 공장은 자재 분배, 부품생산 부분1,2,3, 조립 및 포장 부분과 이들간 물건 운반을 담당하는 AGV로 구성되어 있다. 또, 각 부품생산 부분은 몇 개의 기계들로 구성된다.

2) 모델 구조

위의 시스템을 시뮬레이션 하기 위해 생성한 모델 구조는 <그림 14>에서와 같이 시스템의 구성이 4개의 계층성을 갖는다. 트리 형태의 구조에서 단말노드에 있는 모델들이 기본 모델이고, 그 외의 노드들은 모두 결합 모델이다.



<그림 14> 제조 공정 시스템의 모델구조

각 계층에 해당하는 모델 구성 요소는 다음과 같다.

- Level 1 : Factory
- Level 2 : Instore, AGV1, ShopFloor, AGV3, OutStore
- Level 3 : MateBuff, Manager, /Distributor, AGV2, WC1, WC2, WC3, Comb&Pack, / Deliver, ProdBuff
- Level 4 : M11, M12, M13, / M21, M22, M23, M23, / M31, M32

3) 모델 정의 예

테스트 모델 구조에서 자재 입력담당의 운반 시스템인 기본모델 Manger의 애니메이션 객체의 정의 및 동작 명세와 결합모델인 WorkCell을 나타내는 이미지 객체를 정의한 명세이다. 각 객체 정의에 필요한 그래픽 데이터는 "factory.dat"파일에 있다.

▸ 기본 모델의 애니메이션 객체 명시

[변수]

```
Graphic_Lib graphic;
Img_List * m_ilistGPT;
CDynImage * m_dynTruck;
CDynImage * m_dynPro;
CPoint     m_pStart;
CPoint     m_pEnd;
```

[초기화 함수]

trcuk(char * name,timetypeprocessing_time,
CPoint pStart ,CPoint pEnd)

```
this->processing_time = processing_time ;
m_pStart = pStart;
m_pEnd = pEnd;
m_pJob = NULL;
m_ilistGPT=graphic.Read("factory.dat");
m_dynTruck=new CDynImage();
m_dynTruck ->LoadImage(m_ilistGPT,"truck");
```

[상태변이 함수]

deltext(timetype e,message * x)

```
if(  phase_is("passive"))
{
        m_pJob=(Job*)x->get_val_on_port("in",i);
        hold_in("transfer",processing_time/2);
        ResetAInfo();
        m_dynTruck->SelectDraw(true);
        m_dynTruck->MoveTo(m_pStart);
        A_info(m_dynTruck, m_pEnd,
                (int)processing_time/2);
        m_dynPro->SelectDraw(true);
        m_dynPro->MoveTo(m_pStart);
        A_info(m_dynPro, m_pEnd,
                (int)processing_time/2);
}
```

▸ 결합 모델의 이미지 객체 명시

[모델 정의]

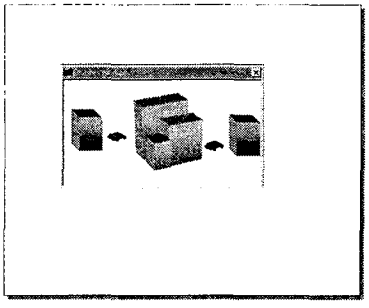
```
digraph * m_digWC1;
```

[배경 이미지 정의]

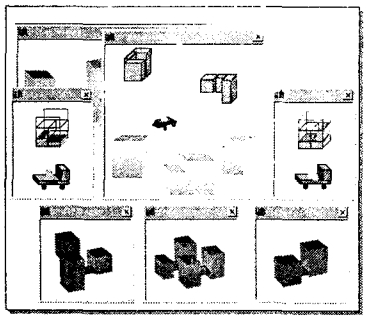
```
Graphic_Lib graphic;
Img_List * ilistGPT;
CImage   * temp;
ilistGPT=graphic.Read("factory.dat");
temp = new CImage();
temp ->LoadImage(ilistGPT,"WorkCell1");
temp->MoveTo(CPoint(100,200));
m_digWC1->I_info(temp);
```

4) 애니메이션 출력

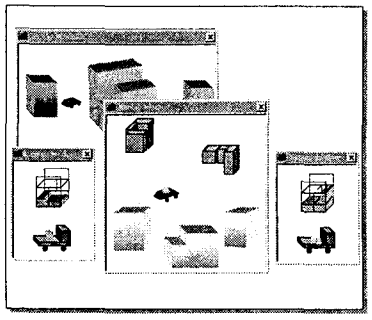
위의 모델구조에 따라 계층적 애니메이션 한 결과는 <그림 15>, <그림 16>, <그림 17>과 같다. <그림 15>는 모델 구조에서 두 번째 계층의 모든 구성 요소를 하나의 출력 창으로 공정 시스템의 들어오는 입력(자재)과 출력(제품)의 상황만을 관찰하고자 하는 출력 형태이다. <그림 16>은 모델 구조에서 세 번째 계층의 모든 구성 요소를 자재 반입, 공정 시스템, 제품 출하를 개별적인 출력 창으로 애니메이션 한 것으로써, 관찰자가 시스템 내부의 상황에 관심이 있는 경우의 출력형태이다.



<그림 15> 제조공정시스템의
두 번째 계층 출력



<그림 16> 제조공정시스템의
세 번째 계층 출력



<그림 17> 제조 공정 시스템의
네 번째 계층 출력

<그림 17>은 공정 시스템의 각 부품 공정의 내부를 나타낸 모델 구조에서 네 번째 계층까지의 애니메이션을 모두 보인 것이다. 그림에서 알 수 있듯 계층적 애니메이션을 통해 관찰자는 원하는 계층 및 구성요소의 시뮬레이션 과정을 직관적으로 이해 할 수 있다.

<그림 16>은 모델 구조에서 세 번째 계층의 모든 구성 요소를 자재 반입, 공정 시스템, 제품 납입별 출력 창으로 애니메이션 한 것이고, <그림 17>은 공정 시스템의 각 부품 공정의 내부를 나타낸 모델 구조에서 네 번째 계층까지의 애니메이션을 모두 보인 것이다. 그림에서 알 수 있듯이 계층적 애니메이션을 통해 관찰자는 원하는 계층 및 구성요소의 시뮬레이션 과정을 직관적으로 이해할 수 있다.

5.3 애니메이션 정확도 고찰

시뮬레이션을 표현하는 애니메이션의 정확도는 모델의 이벤트 스케줄링을 얼마나 애니메이션 동작에 반영하는지에 따라 평가될 수 있다. 화면에 그래픽 이미지를 출력하는 시간의 흐름은 컴퓨팅 시스템의 상태에 따라 다를 수 있으므로, 모델별 애니메이션 객체들의 움직임이 현실의 시간 흐름에서 정확히 시뮬레이션 시간 비율로 애니메이션을 수행한다고 보장 할 수 없다. 하여 이벤트 전후간에 애니메이션 정확도는 알고리즘에 의존 할 수밖에 없다.

4장에서 설계한 계층적 애니메이터 알고리즘에 의한 애니메이션의 정확성은 시뮬레이션 모델 전체의 이벤트 단위까지 보장한다. 즉, 한 모델이 애니메이션 동작 할 때 발생하는 그래픽적 또는 시간적 에러는 누적되지 않으므로 다음 이벤트의 새로운 애니메이션 동작에 영향을 끼치지 않는다. 이는 애니메이터가 이벤트 스케줄링의 선행관계가 애니메이션 선행관계로 반영되도록 시뮬레이터와 결합하여 함께 시뮬레이션 사이클 주기를 만드는 기틀을 마련해 주기 때문이다.

6. 결론

신뢰성 있는 시뮬레이션을 하는데 많은 어려움이 개발자의 모델코드 검증, 모델내용 검증, 대상 시스템 전문가의 시뮬레이션 이해의 어려움 등에서 야기된다. 이러한 어려움을 개선하기 위한 방법의 하나로 시뮬레이션 툴 개발자들은 시뮬레이션의 동적인 특징을 효율적인 애니메이션 형태로 출력하는 것에 관심을 갖는다.

본 연구에서는 시뮬레이션 대상 시스템의 동적인 변화를 원하는 계층에서 관찰할 수 있도록 하는 계층적 애니메이션을 제안하여 전체적인 개발환경을 설계하고 시뮬레이션과 애니메이션의 연결 부분인 계층적 애니메이터 구현을 통해 검증하였다. 계층적 애니메이션을 통해 개발자 및 시스템 전문가들은 시스템의 변화를 계층 및 요소별로 관찰 할 수 있고, 이는 시뮬레이션의 이해를 용이하게 함으로써 시뮬레이션의 신뢰성 검증을 향상시키는 효과를 기대할 수 있다. 또한, 애니메이션 출력을 모델 단위로 할 수 있어 모델 구조의 재구성을 용이하게 하며 한정된 화면으로도 효과적인 애니메이션을 출력 할 수 있게 한다.

향후 연구 내용으로는 진행된 연구가 알고리즘 위주의 설계 및 구현 검증이었던 만큼 그래픽적 기능이 부족하여 이에 대한 보완이 필요하고, 이벤트 단위 모델 구조와 같은 구조의 추상 애니메이터가 생성되어 각 모델이 시뮬레이션부터 애니메이션까지의 과정이 독립적으로 동작함을 고려할 때 계층적 애니메이션을 분산 환경의 출력에 적용 할 수 있다.

참고 문헌

[1] David R. Hill, Object-Oriented Analysis and Simulation, Addison-Wesley, 1996.

[2] Bernard P. Zeigler, Object-Oriented Simulation with Hierarchical, Modular Models, Academic Press, 1990.

[3] Averill M. Law and W. David Kelton, Simulaiton Modeling and Analysis, Second Edition, McGraw-Hill, 1991.

[4] Bernard P. Zeigler, Object and Systems, Springer-Verlag,1997.

[5] Bernard P. Zeigler, Theory of Modeling and Simulation , John Wiley, 1976, reissued by Krieger, Malabar,1985.

[6] Bernard P. Zeigler, Multifacetted Modeling and Discrete Event Simulation, Academic, 1984.

[7] Lan Sommerville, Software Engineering, Fifth Edition, Addison -Wesley, 1995.

[8] Paul A. Fishwick, Simulation Model Design and Execution, Prentice Hall, 1995.

[9] Hyup J. Cho, DEVS-C++ Reference Guide, 1997.

[10] CACI Company, MODSIM III Manual, 1997.

[11] David J. Kruglinski, Inside Visual C++, Microsoft Press, 1996.
[12] Tae H. Cho, A Hierarchical, Modular Simulation Environment for Flexible Manufacturing System Modeling , Ph. D. dissertation, Univ. Of Arizona, Tucson, 1993.
[13] Nadia M. Thalmann, Daniel Thalmann, Computer Animation, ACM Computing Surveys, Vol.28, No.1, March 1996.
[14] Siroos Sokhan-Sanj ,Gerald T. Mackulak, The value of Simulation Animation: Discussion of Instances Where Statistical output is Insufficient for Analysis of System Performance , Proceeding of the 2nd Annula International Conference on Industrial Engineering Application and Practice II, Vol. 2, 1997, Califonia USA.
[15] Paul A. Fishwick, Hanns-Oskar A. Porr, Using Discrete Event Modeling For Effective Computer Animation Control, Winter Simulation Conference, Dec. 1991, Phoenix, AZ, pp.1156-1164.
[16] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips, "Introduction To Computer Graphics" , Addison-Wesley, 1994.
[17] Yi-Bing Lin, Edward D. Lazowska, "A Study of Time Warp Rollback Mechanisms", ACM Transactions on Modeling and Computer Simulation, Vol.1, No.1, 1991.
[18] 조대호, 이칠기, "계층의 구조를 갖는 시뮬레이션 모델에 있어서 단계적 접근을 위한 모델연결 방법론과 그 적용 예" , 한국시뮬레이션학회 논문지, 제5권, 제2호, 1996.
[19] 조대호, 이미라, "시뮬레이션의 계층적 애니메이션 환경" , 한국시뮬레이션학회 학회지, 추계학술대회, 1998.

저자소개

이미라
1998 성균관대학교 정보공학과 학사
1998~현재 성균관대학교 전기전자 및 컴퓨터공학부 석사과정
관심분야: 이산사건 시뮬레이션, 시뮬레이션 개발 환경, 인공지능

조대호
1983 성균관대학교 전자공학과 학사
1987 알라바마대 전자공학과 석사
1993 아리조나대 전자 및 컴퓨터공학 박사
1993~1995 경남대학교 전자계산학과 전임강사
1995~현재 성균관대학교 전기전자 컴퓨터공학부 부교수
관심분야: 모델링 시뮬레이션, 공장 자동화, 지능 제어, ERP