AOP 기술을 이용한 DEVS 기반 시뮬레이터의 적응성 향상 기법

Improving Adaptability of the DEVS Based Simulator with AOP

김 철 호*
Kim, Cheol-Ho

ABSTRACT

The DEVS formalism has the well-defined relationship between its model and simulator. However, it does not define the connection between its model and experimental frame needed when a simulator's implemented with it. So, in most DEVS based simulators, the modules of the two parts are tangled, so that changing and reusing them is not easy. This paper proposes a method to improve the changeability of the experimental frame and the reusability of the model by modularizing the two parts using the AOP technology. I applied the new method to a real project, and the result shows that it improves the two qualities effectively than before.

주요기술용어(주제어) : Ship Combat System(함정전투체계), M&S(Modeling and Simulation), DEVS(Discrete Event Systems Specifications), AOP(Aspect-Oriented Programming), Adaptability(적응성), Reusability(재사용성), Changeability(변경용이성)

1. 머리말

M&S(Modeling and Simulation)를 이용하여 효율적인 개발을 수행하고 고품질의 결과물을 얻고자 하는 시도가 최근 주목을 받고 있다. 이 경향은 함정전투체계(Ship Combat System) 분야에서도 다르지 않다. 전투체계의 다양한 요소들을 모델링하고 시뮬레이터를 이용해 실험한 후, 그 결과를 체계개발에 반영하고 있다.

† 2007년 3월 23일 접수~2007년 6월 15일 게재승인
* 국방과학연구소(ADD)
주저자 이메일 : aceman@add.re.kr

전투체계는 신호처리를 중심으로 하는 연속 시스템(Continuous System)과 달리, 임의의 시간에 사건이 불규칙으로 발생하고 각 사건들이 시스템의 상태에 영향을 주는 이산 시스템(Discrete System)의 일종으로 분류될 수 있다. 이산 시스템을 효과적으로 모델링할 수 있는 방법이 있다면, 전투체계를 모델링하는데도 그 방법이 가장 적합하다고 할 수 있다.

B. P. Zeigler 교수가 제안한 DEVS(Discrete Event Systems Specifications)[1]는 이름에서 알 수 있듯이 이산시스템을 기술하기 위해 고안된 모델 명세법으로써 크게 두 가지 특징을 가진다. 첫째, 집합론을 기반으로 하는 형식론으로 그 수학적 엄밀성이 모델의 V&V(Verification and Validation)를 용이하게

해 준다. 둘째, 이산사건 모델을 계층적이고 모듈러(modular)하게 표현할 수 있는 체계를 제공한다. 즉, 복잡한 시스템을 각 구성요소 별로 나누어 모델을 만든 후, 이를 합쳐서 전체 시스템을 표현할 수 있도록 되어 있다[2,3]. 이러한 장점으로 전투체계 뿐 아니라 여러 국방관련 프로젝트에 DEVS가 적용되고 있다.

DEVS는 추상화 시뮬레이터[2]를 정의하여 모델과 시뮬레이터 사이를 분리함으로써, 모델러(modeler)가 모델링에만 전념할 수 있도록 해 줄 뿐 아니라, 소프트웨어 품질면에서 우수한 시뮬레이터를 구현할 수 있게 하였다. 그러나, DEVS는 실험틀(experimental frame)과 모델의 관계는 정의하지 않는다. 실험틀과 모델의 관계를 구현하는 기존의 방법은 이 둘 사이에 강한 의존성(high dependency)을 유발한다. 즉, 실험틀에 대한 변경을 위해 모델 내부를 변경해야 하거나, 실험틀 관련 코드로 인해 모델의 재사용성(reusability)이 저해되는 문제가 있다. 전체적으로 시뮬레이터의 적응성(adaptability)[4]이 낮아진다.

최근에 들어 시뮬레이션 학계에서 실험틀과 모델의 의존성을 최소화하여 실험틀의 적응성을 향상시키려는 시도가 시작되었고[5,6], 그 적용 기술로 AOP (Aspect Oriented Programming)[7]을 고려하는 연구가 있었다[8,9]. 하지만, 이 연구들은 DEVS가 아닌 다른 명세법을 대상으로 하고 있고, 구현 수준에서 아이디어 또는 디자인을 제시하는 단계이거나 현재 최초 버전을 내놓은 상태이다. 또 구현을 위해 사용한 언어도 성능 등의 이유로 선호되는 C++보다 자바(Java)를 사용하고 있거나 구체적인 언어가 명시되고 있지 않다. 이런 이유로 DEVS와 C++ 기반의 시뮬레이터에는 이 연구들이 제안하는 방법을 바로 적용하기는 어렵다. 이에 상기한 조건의 시뮬레이터에 바로 적용할 수 있는 AOP 기반의 적응성 향상 기법을 고안하게 되었다.

이 논문에서는 기 제작된 시뮬레이터의 개선 프로젝트를 통해 고안한 기법을 제안하고, 그것을 적용한 과정과 효과를 분석해서 제시한다. 이후로 이 논문은 배경 지식, 기존 구현의 문제점 설명, 향상 기법 제시, 프로젝트 소개, 실제 적용 결과 및 효과 분석의 순서로 진행된다.

2. 배경 지식

가. DEVS와 DEVSimHLA

DEVS 형식론은 집합 이론에 기반하여 이산사건 시스템을 개발하기 위한 시뮬레이션 이론으로, 3개의 집합(입력집합(input set), 출력집합(output set), 상태집합(state set))과 4개의 함수(external transition function, internal transition function, output function, time advanced function)를 이용해 시스템을 분석하고 설계하기 위한 틀을 제공한다. 이를 통해 개발자는 시뮬레이터의 구현을 위한 모델의 완전성을 확보할 수 있게 되고, 모델의 확장성 및 유지보수성을 높일 수 있게 된다[1~3].

DEVS는 시스템을 표현하기 위해 원자 모델(Atomic Model, 이하 AM)과 결합 모델(Coupled Model, 이하 CM)의 두 가지 형태의 모델을 제시한다. AM은 시스템에서 사용되는 각 컴포넌트들의 기능을 표현해 주고, CM은 시스템 전체를 계층적/구조적으로 표현할 수 있는 기능을 제공한다. DEVS 모델의 전체적인 흐름은 외부로부터 입력을 받으면 현재의 상태와 조건에 따라 출력을 내보내고 현재의 상태를 바꾸어 시스템을 변화시킨다. 그리고 현재 상태에서 머물 수 있는 시간을 지정하여 지정된 시간이 경과하게 되면 모델 스스로 현재의 상태를 변화시키게 된다. 세부적인 AM의 명세는 다음과 같다.

AM = <X, Y, S, δ_{ext}, δ_{int}, λ, ta>

- X는 입력값의 집합이다.
- Y는 출력값의 집합이다.
- S는 상태들의 집합이다.
- δ_{ext} : Q × X → S는 외부 상태 천이 함수(external transition function)이고, Q는 전체 상태의 집합인데 다음과 같이 정의된다.

  $Q = \{(s, e) | s \in S, 0 \le e \le ta(s)\}$

  여기서 e는 마지막 상태 천이 후 흐른 시간을 말한다.
- δ_{int} : S → S는 내부 상태 천이 함수(internal transition function)이다.
- λ : S → Y는 출력함수(output function)이다.
- ta : S → $R^{+}_{0,\infty}$은 시간전진함수로서 0과 ∞사이의 양의 실수(實數)값을 가진다.

AM의 동작은 다음과 같은 규칙을 따른다.

(1) AM은 특정 시간에 하나의 상태에만 머물 수 있다.

(2) 외부 입력 없이 하나의 상태에 머물 수 있는 시간은 시간전진함수 ta(s)에 의해 결정된다.

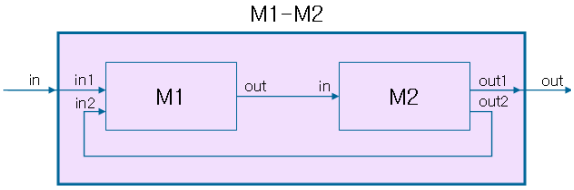
(3) AM의 상태($0 \leq e \leq ta(s)$)는 외부 입력에 의한 실행함수인 δ_{ext}에 의해 변화된다.

(4) 하나의 상태에서 머물 수 있는 시간이 지나면 출력함수(λ)를 생성하고 δ_{int}에 의해 현재의 상태가 변화된다.

CM의 명세는 예를 들어 설명하겠다. 그림 1은 DEVS형식론의 CM을 보여준다.

M1과 M2는 DEVS 형식론의 AM이다. M1은 “in1”과 “in2”의 두 개의 입력을 가지고 “out”이라는 하나의 출력을 가진다. M2는 “in1”의 하나의 입력을 가지고 “out1”과 “out2”의 두 개의 출력을 가진다. M1의 출력(“out”)은 M2의 입력(“in”)과 연결되어 있으며 이를 IC(Internal Coupling)이라 한다. 또, M1의 입력(“in1”)은 CM인 M1-M2의 외부에서 들어오는 입력(“in”)과 연결되어 있는데 이것을 EIC (External Input Coupling)라 한다. 마지막으로 M2의 출력(“out1”)은 M1-M2의 전체의 출력(“out”)과 연결되어 있는데 이를 EOC(External Output Coupling)라 한다.

DEVSimHLA[2]는 C++에 기반한 시뮬레이션 엔진으로 Microsoft Visual Studio .NET으로 개발할 수 있는 시뮬레이션 라이브러리를 제공한다. DEVSimHLA를 이용하면, 시뮬레이션 엔진 자체에서 시뮬레이터 전체의 시간 관리를 자동으로 해주므로, 개발자는 시뮬레이션 전체의 시간 관리를 위한 코딩을 일일이 해줄 필요가 없으며, 제공된 라이브러리를 이용해 쉽게 시뮬레이터를 개발할 수 있는 장점을 가지게 된다. 이 논문에서는 DEVSimHLA을 DEVS 모델 구현 및 시뮬레이션 개발 환경으로 사용하였다.



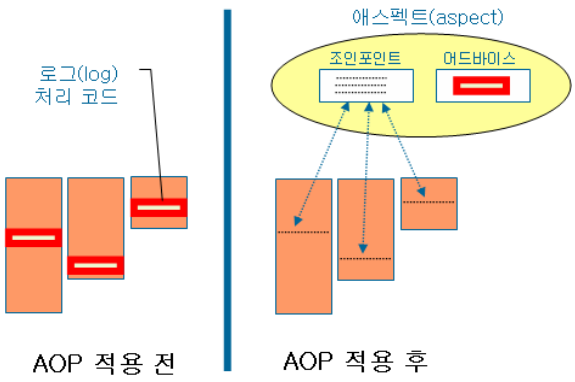
[그림 1] DEVS 결합 모델

나. AOP와 AspectC++

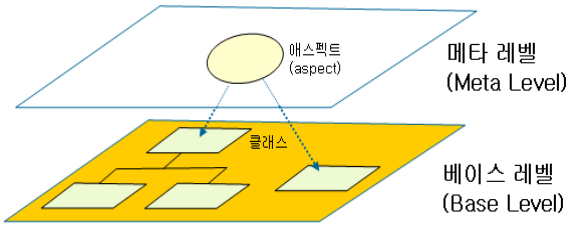
AOP는 특정한 소프트웨어 개발 방법(예, 객체지향 방법론)을 사용했을 때, 여러 모듈에 흩어져서 구현되는 요구사항/기능을 모듈화시키는 기술이다. 이런 요구사항/기능을 횡단관심(crosscutting concern)이라고 부르는데, 로깅(logging), 보안(security), 오류처리(error handling) 등이 횡단관심에 속한다고 할 수 있다.

그림 2는 여러 모듈에 흩어져 있는 로그 처리 코드를 애스펙트(aspect)로 모듈화하는 예제인데, AOP 언어가 가지는 일반적인 요소(construct)들을 잘 보여주고 있다. 애스펙트는 크게 두 요소로 구성되는데, 조인포인트(joinpoint)와 어드바이스(advice)가 그것이다. 조인포인트는 애스펙트가 동작해야 할 시점을 정의하고, 어드바이스는 해당 시점에서 실행해야할 기능을 정의한다. 조인포인트와 어드바이스는 모두 애스펙트 내에 정의된다. 예제에서는 로그 처리 코드들이 애스펙트의 어드바이스로 구현되고, 로그 처리가 이루어져야 할 시점이 조인포인트로 선언된다.

AOP 기술의 큰 특징은 대상 시스템이 존재하는 레벨(level)보다 상위 레벨에서 애스펙트가 구현된다는 것이다. 애스펙트가 구현되는 레벨을 메타 레벨(Meta Level)이라 부른다. 그림 3에 나타나 있는 것과 같이 애스펙트는 메타 레벨에서 구현되고, 대상 시스템을 구성하는 클래스들은 베이스 레벨(Base Level)에서 구현된다. 메타 레벨에서는 베이스 레벨



[그림 2] 애스펙트 예제



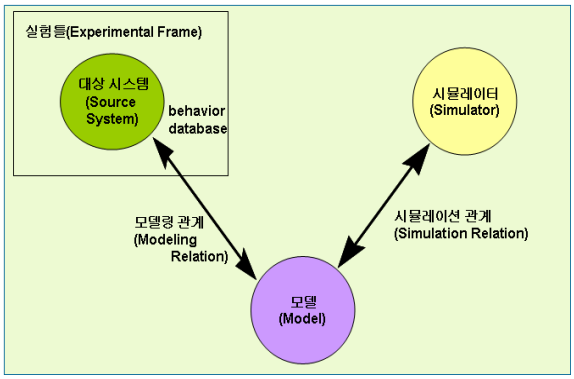
[그림 3] AOP 기술의 구현 방식

에 있는 모든 요소에 접근할 수 있다. 이런 구현 방식으로 얻을 수 이점은 두 가지이다. 첫째는 베이스 레벨에서 여러 클래스에 산재되어 있는 모듈 또는 코드를 모듈화하기가 용이하다는 것이다. 둘째는 베이스 레벨의 클래스 또는 모듈 사이의 의존성을 낮추는 것은 물론이고, 의존성역전원리(Dependency Inversion Principle)를 적용하기도 용이하다는 것이다.

AspectC++[10,11]는 C++을 확장한 일반적인 목적의 AOP 언어이다. 기존에 AspectJ[12]라는 Java를 확장한 언어가 있지만, 실행 효율성(runtime efficiency) 등의 요구를 만족시기 위해 AspectJ의 프로그래밍 스타일을 C++에서 지원하도록 하는 것이 AspectC++의 목적이다. AspectJ의 영향을 받았지만 AspectC++은 C++ 영역에 유일한 개념들도 지원한다. 예를 들면, 연산자 오버로딩, 다중 상속 등이다. 현재 GPL (GNU General Public License)하에 소스 코드까지 배포되고 있다. AOP를 지원하는 언어나 프레임워크가 다양하게 존재하지만, 본 논문에서는 대상으로 하는 시뮬레이터가 C++를 기반으로 작성되어 있어 AspectC++를 채택하였다.

3. DEVS 기반 시뮬레이터의 구현상 문제점

DEVS는 모델 명세 이론 뿐 아니라, 이것을 실제 소프트웨어로 구현할 때 필요한 프레임워크와 알고리즘도 정의한다. 그림 4는 DEVS가 제안하는 M&S 프레임워크를 나타내고 있는데, 모델, 시뮬레이터, 실험틀로 구성된다. 모델은 대상 시스템을 모델링한 결과를 말하며, 시뮬레이터는 모델을 실행할 수 있는 환경을 뜻하고, 실험틀은 시나리오 생성, 실시간 모니터링, 후처리용 데이터 수집 및 결과 분석 등 시뮬레



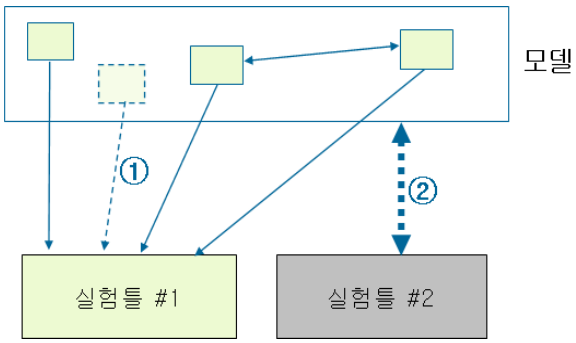
[그림 4] M&S 프레임워크[1]

이션 관리를 위한 각종 기능을 제공한다. DEVS는 또한 특정 모델에 귀속되지 않는 추상 시뮬레이션 알고리즘을 정의하는데, 이를 시뮬레이터에 구현함으로써 모델의 실행은 시뮬레이터가 모두 담당하게 된다. 모델러(modeler)는 모델의 실행 제어 등을 고려하지 않고 대상 시스템의 행위 자체를 모델링하는데만 몰두할 수 있다.

소프트웨어 품질 관점에서 보면, 이런 방식을 통해 모델과 시뮬레이터 사이의 의존성을 최소화시켜 두 구성요소의 변경용이성(changeability)과 재사용성이 획기적으로 향상되었다고 볼 수 있다. 이 논문에서는 변경용이성과 재사용성을 시뮬레이터의 적응성으로 정의한다. 적응성은 소프트웨어 품질의 하나로써, 대상 시스템에 부여된 제약조건(constraint)이나 사용자 요구의 변화에 대해 소프트웨어가 얼마나 쉽게 대응할 수 있는지에 대한 정도를 나타낸다.

이에 반해 DEVS는 실험틀과 모델 사이의 관계에 대해서는 구현 수준에서 정의하지 않는다. 실험틀과 모델의 관계를 구현하는데 사용하는 전형적인 방법은 DEVS 모델 내에 특별한 모델과 모듈을 추가로 만들고 그 모델과 실제 모델 사이를 연결시켜 실험틀의 기능을 지원하도록 하는 것이다. 이 방법은 모델의 재사용성과 실험틀의 변경용이성을 저해하게 된다. 이 문제를 그림 5를 통해 설명하겠다.

그림 5의 상단에 있는 사각형이 모델이며 아래에 두 개의 사각형이 각각 실험틀을 나타낸다. 앞에서 설명했던 바와 같이 보통 실험틀 #1과 모델이 서로 연동하기 위해서는 모델 내에 실험틀 #1을 지원해 주



[그림 5] 기존 구현 방식의 문제점

는 모델과 모듈을 작성한다. 이 경우 만일 실험틀 #1에서 수집해야 할 데이터가 추가될 경우에는 모델 내에 이것을 지원할 모듈이 추가되어야 한다. 그림 5의 ①이 새로운 모듈과 실험틀 #1과의 연결관계를 나타낸다. 이런 방식이 가지는 문제 중 첫번째는 실험틀의 변경용이성을 저해된다는 것이다. 즉, 실험틀 #1에 대한 요구 사항의 변화가 실험틀 #1의 변경 뿐 아니라, 실험틀의 기능과 관련이 없는 모델의 변경(내부 모델 또는 모듈의 변경)을 유발하기 때문이다. 또 다른 문제는 모델의 재사용성도 낮다는 것이다. 만약 새로운 실험틀 #2가 모델을 재사용하려고 할 경우, 모델 내의 지원 모듈은 실험틀 #1을 위한 것이므로 이들을 모두 변경 또는 삭제해야 실험틀 #2에서 모델을 사용할 수 있다. 즉, 모델의 재사용 시 모델 내부의 변경이 불가피하다.

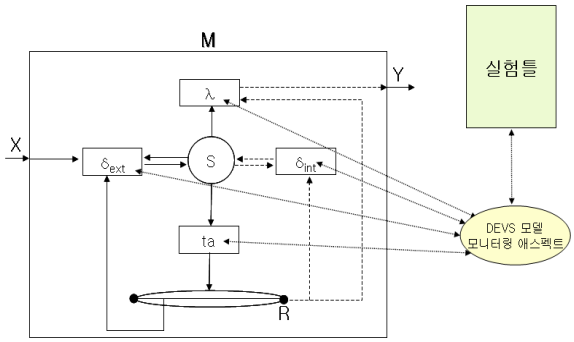
만약 모델 내부의 실험틀 지원 기능들을 모듈화하여 모델 밖으로 분리시킬 수 있으면서 효율적으로 모델의 실행을 모니터링(monitoring) - 활동을 감시하고 데이터를 수집 - 할 수 있다면 앞에서 지적한 두 가지 문제가 해소될 수 있겠다. 이 논문은 두 가지 문제를 해결할 수 있는 방법과 실제 적용한 사례 및 효과를 제시한다.

4. AOP 기반의 적응성 향상 기법

실험틀과 모델 간의 문제를 해결하기 위해서는, 먼저 모델의 행태 분석을 통해 모니터링 시점과 수집해야할 데이터를 이해하는 것이 중요하다. DEVS의 모델은 두 종류가 있지만 원자 모델에서만 실제 동작이 기술된다. 그림 6에서 "M"은 2장에서 설명한 DEVS 원자모델의 동작을 그림으로 나타낸 것이다. 4개의 함수가 사각형으로 표현되어 있다.

모델에 대한 모니터링은 해당 모델의 동작을 감시하다가 특정 시점에서 필요한 정보를 수집하는 것이라 할 수 있다. 이산사건모델은 사건에 의해서 그 상태가 변하게 되고, 해당 모델의 상태가 변하는 시점이 내부 데이터와 외부 출력이 변경되는 시점이다. 그 외 시점에서는 내부 변경이나 외부 출력이 발생하지 않는다. DEVS 모델에서는 상태 변경이 2개의 함수로, 결과출력이 1개의 함수로, 그리고 시간전진이 1개의 함수로 표현된다. 모델의 행위가 이 4개의 함수로 이루어지므로, 4개의 함수의 실행 시점을 감시할 수 있으면 모델의 동작을 모니터링할 수 있다. 또, 모니터링 데이터는 모델 상태 변경시 모델 내부에 접근할 수 있다면 상태 정보나 실행결과값 등의 필요한 데이터를 수집할 수 있다.

이 논문에서는 DEVS 모델에 대한 효율적인 모니터링을 위해 그림 6과 같이 원자 모델 및 실험틀과 AOP의 애스펙트를 결합시키는 방법을 제안한다. 이 방법은 기존의 DEVS 원자 모델과 실험틀 외에 모니터링을 위한 애스펙트를 정의한다. 이 애스펙트를 DEVS 모델 모니터링 애스펙트(DEVS Model Monitoring Aspect, 이하 DMMA)라고 명명하였다. DMMA는 원자모델의 4개 함수를 조인포인트로 정의하고 데이터 수집과 실험틀에 전송하는 부분은 어드바이스로 구현한 애스펙트이다. 이렇게 함으로써 모델의 상태가 변경되는 시점을 DMMA를 통해 감시할



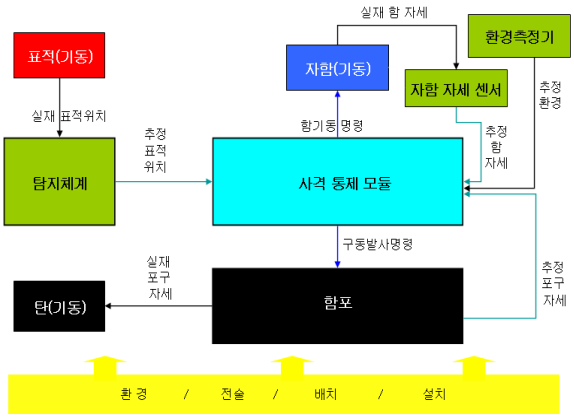
[그림 6] DEVS 원자 모델[3]과 애스펙트와의 결합

수 있다. 또, DMMA는 메타레벨에서 모델의 내부에 접근 가능하므로 데이터 수집에도 문제가 없다. 정확한 시점에 필요한 데이터를 수집한 DMMA는 이것을 실험틀에게 넘겨주는 것으로 그 역할을 마무리한다.

이 방식에 따르면 실험틀은 모델에 모니터링과 관련된 모델이나 모듈을 추가하지 않고 DMMA를 이용해서 모델을 감시할 수 있다. 이 방식으로 얻을 수 있는 효과는 다음과 같다. 첫째, 실험틀과 모델은 DMMA를 기준으로 분리되어 각각의 변경의 영향은 상대편으로 전파되지 않는다. 둘째, DMMA내에 모델과 실험틀의 연관 관계가 정의되어 있고 모델과 실험틀 내부에는 상대에 관련된 모듈이 없어, 모델과 실험틀 모두 재사용시 서로 의존적이지 않다. 즉, 제안한 기법을 적용할 경우, 그림 5의 구조에 비해 실험틀의 변경용이성과 모델의 재사용성이 향상된 구조를 얻을 수 있다.

5. 프로젝트 소개

이 논문은 함정의 함포중심 교전 시뮬레이터(Gun-oriented Engagement Simulation System)[13]의 개선 프로젝트를 통해 시작되었다. 함포중심 교전 시뮬레이터란 함포의 명중률을 평가하고 그 명중률에 영향을 미치는 여러 가지 요소들을 평가하기 위한 시뮬레이션 도구이다. 그림 7은 이 시뮬레이션의 개념적 구성도이다. 일반적인 함정의 사격통제 시스템 및 각



[그림 7] 함포중심 교전 시뮬레이터 구성도

하부 시스템과 이들의 연동관계를 보여주고 있다.

개선 프로젝트는 다음의 세 가지 요구 사항을 달성하는 것을 목표로 한다.

(1) 모델과 GUI(전황전시기 및 시나리오 발생/관리기)을 서로 독립적으로 변경 가능 하도록 분리시킬 것

(2) 시뮬레이션의 분석 대상(모델 또는 수집 데이터)의 변경이 용이 할 것

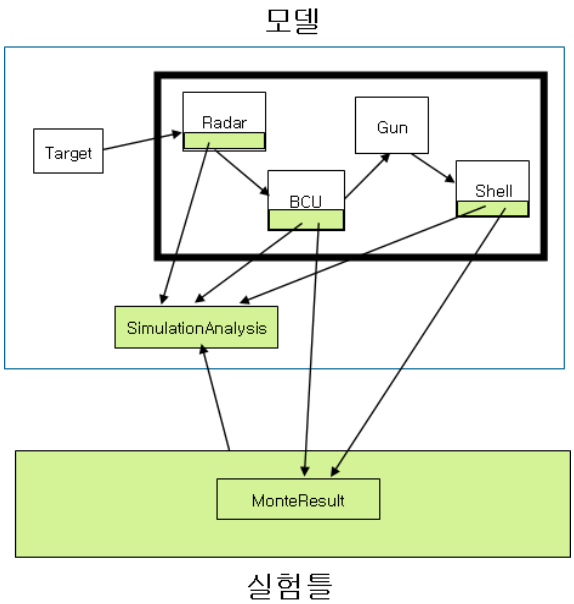
(3) 새로운 모델(예, 다대다 교전 모델)설계 시 기존 모델의 재사용이 용이할 것

6. 시뮬레이터의 기존 구조

그림 8은 개선 프로젝트 수행 전의 함포중심 교전 시뮬레이터의 구조를 나타낸다. 여기에는 모니터링 관련 모델과 클래스가 있다. SimulationAnalysis 모델은 일반 시뮬레이션 수행시 데이터를 수집한다. MonteResult 클래스에는 몬테칼로 시뮬레이션 시에 수집된 데이터가 저장된다. 이 구조는 모델과 실험틀의 관계에서 발생할 수 있는 전형적인 문제점 두 가지를 보여주고 있다.

기존 구조에는 실험틀이 필요로 하는 데이터를 수집하기 위해서는 이 일을 전담하는 SimlationAnalysis 모델이 필요하다. 그리고 Radar, BCU, Shell 모델내에는 필요시 데이터를 SimlationAnalysis에게 전달하는 모듈과 SimlationAnalysis과의 연결관계를 정의해야 한다. 만일 요구사항이 변경되어 실험틀에서 Gun을 모니터링해야 한다면, SimulationAnalysis과 Gun 사이의 관계와 Gun 모델 내에 지원 모듈을 추가해야 한다. 즉, 실험틀의 변경이 모델의 변경을 유발한다. 이것은 실험틀의 변경용이성에 대한 것으로 첫 번째 문제이다. 또, Radar, BCU, Shell 모델을 재사용하기 위해서는 각 모델 내부에 있는 모듈과 SimlationAnalysis와의 관계를 제거해야 한다. 즉, 모델의 재사용이 실험틀에 의해 제한을 받는다. 이것은 모델의 재사용에 대한 것으로 두 번째 문제이다.

더구나 기존 구조에서는 개발편의성을 위해 모델에서 직접 실험틀의 클래스인 MonteResult에 접근하고 있다. 이런 방식은 모델의 실험틀에 대한 의존성을



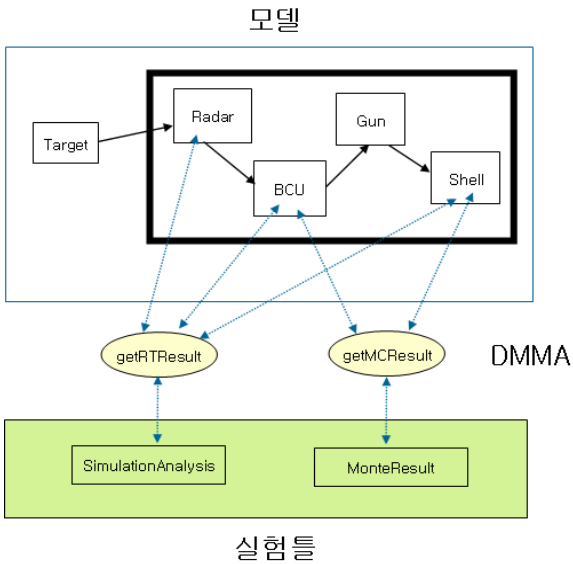
[그림 8] 기존 시뮬레이터의 모델과 실험틀의 관계

강화한다. 실험틀이 수행해야 할 시뮬레이션의 컨텍스트(context)를 반영하고, 모델은 대상 시스템을 반영한다는 관점에서 보면, 모델의 실험틀에 대한 의존성은 바람직하지 못하다.

개선 프로젝트의 요구사항 관점에서 이 세 가지 문제점을 끼치는 영향을 살펴보면, (1) 모델과 GUI의 변경은 독립적이지 못하고 심지어 GUI의 변경 때문에 모델의 변경해야 하며, (2) 분석 대상을 변경하려면 기존 모델과 대상 모델에 실험틀에 의존적인 모듈을 변경/추가해야 하며, (3) 모델에 실험틀 의존적인 모듈이 있어서 모델의 재사용시 변경 작업을 해야 한다. 즉, 개선프로젝트의 목표를 달성하기 위해서는 앞의 세 가지 문제점을 해결하는 것이 중요하다는 것을 알 수 있다.

7. 적용 결과

그림 9는 4장에서 제안한 기법을 적용하여 기존 시뮬레이터의 구조를 변경한 결과이다. getRTResult, getMCResult는 DMMA이다. 모델 내부에 있는 실험틀 관련 모듈이 모두 getRTResult, getMCResult



[그림 9] AOP 기법을 적용한 모델과 실험틀의 관계

로 이동하였다. getRTResult는 일반 시뮬레이션을, getMCResult는 몬테칼로 시뮬레이션을 모니터링한다. 두 개로 나눈 이유는 필요에 따라 착탈을 용이하게 하기 위해서이다. 두 DMMA는 관심 대상인 모델들을 감시하다가 필요한 데이터를 실험틀에 전달한다. 그리고, 기존 구조에서 모델이었던 SimulationAnalysis는 실험틀의 클래스가 되었다.

이 구조가 6장에서 지적한, 기존 구조의 문제점을 모두 해결한다는 것을 구체적으로 기법 적용 전과 후의 코드를 비교함으로써 설명하겠다. 제한된 지면으로 인해 각 경우마다 모델과 DMMA의 코드 일부만을 나타내었다.

첫째, 위 구조에서는 실험틀의 모니터링 기능에 대한 요구사항 변경이 모델의 변경을 유발하지 않는다. 실험틀의 변경용이성이 좋아졌다고 할 수 있다. 예를 들어, Gun 모델의 원자 모델인 GunMotionCalc 모델(함포 자세 계산)이 모니터링 대상 모델로 추가되고 실험틀이 최신의 함포 자세를 수집해야 한다고 가정하자. 기법 적용 전에는 그림 10과 같이 GunMotionCalc 모델의 OutputFn(출력함수)에 SimlationAnalysis로 최신의 함포 자세를 전송하는 코드를 추가해야 한다. 추가된 코드는 이탤릭체로 표시되어 있다. new_gun_motion이 최신 함포 자세 정보를, SimlationAnalysis_

< GunMotionCalc 모델 코드 >

```
#include "..."
            :
bool CGunMotionCalc::OutputFn(CMessage& message)
{
            :
  else if(m_Status == ScanGunMotion)
  {
            :
      message.SetPortValue("OUTPUT_PORT", new_gun_motion);
      message.SetPortValue("SimlationAnalysis_PORT",
                                          new_gun_motion);
  }
  return true;
}
```

[그림 10] 기법 적용 전 GunMotionCalc 모델 코드 변경

PORT는 SimlationAnalysis으로 자료가 전송되는 포트(port)를 의미한다. 그리고 포트를 통한 자료 전송을 위해서는 GunMotionCalc 모델을 포함하는 결합 모델들의 코드도 변경되어야 한다. 지면의 제한으로 결합 모델들의 코드는 나타내지 않았다.

기법 적용 후에는 모델의 코드는 변경되지 않는다. 그림 11과 같이 단지 getRTResult에 GunMotionCalc 모델의 OutputFn을 조인포인트로 하는 새로운 어드바이스를 추가하면 된다. 이 어드바이스는 GunMotionCalc 모델이 최신의 자세 정보를 생성하여 출력할 때 마다 이 정보를 SimulationAnalysis에 저장한다. ref_SimlationAnalysis는 SimulationAnalysis 객체의 참조자(reference)를 말하고, gunMotion는 최신의 자세 정보가 저장되는 SimulationAnalysis의 멤버 변수를 말한다.

둘째, 모델 내부에 실험틀 관련 코드가 없다. 그러므로, 모델의 재사용이 실험틀에 의존적이지 않다. 그림 12와 그림 13은 기법 적용 전과 후의 Shell 모델 코드를 보여준다. 그림 12에서 ExtTransFn(외부 상태 천이 함수) 내의 이탤릭체로 된 코드는 모니터링을 위한 코드로써, 직접 실험틀의 MonteResult에 접근해서 탄의 CPA(Closest Point of Approach) 정보를 저장한다. 다른 시뮬레이터에서 Shell 모델을 재사용할 경우에는 이 코드를 제거해야 한다. 하지만, 기법 적용 후인 그림 13에서는 이와 관련된 코드가

< GunMotionCalc 모델 코드 >

```
#include "..."
            :
bool CGunMotionCalc::OutputFn(CMessage& message)
{
            :
  else if(m_Status == ScanGunMotion)
  {
            :
      message.SetPortValue("OUTPUT_PORT", new_gun_motion);
  }
  return true;
}
```

< getRTResult 코드 >

```
            :
advice  execution("%  CGunMotionCalc::OutputFn(...)")  &&
that(gunmotioncalc) : after (CGunMotionCalc& gunmotioncalc) {
    if (gunmotioncalc.m_Status == gunmotioncalc.ScanGunMotion) {
    {
        ref_SimlationAnalysis.gunMotion
                                = gunmotioncalc.new_gun_motion
    }
}
```

[그림 11] 기법 적용 후 getRTResult 코드 변경

< Shell 모델 코드 >

```
#include "..."
            :
bool CShell::ExtTransFn(const CMessage& message)
{
            :
  if((cmd == "FireShell") && (m_Status == Wait)){
  {
            :
      ProjTraj_calculate(...);
      ref_MonteResult->SetCPA(...);
            :
  }
  return true;
}
```

[그림 12] 기법 적용 전 Shell 모델

getMCResult의 어드바이스로 구현되어 있으며, Shell 모델에는 더 이상 실험틀과 관련된 코드가 없다. 모델의 재사용성이 좋아졌다고 할 수 있다.

< Shell 모델 코드 >

```
#include "..."
                :
                :
bool CShell::ExtTransFn(const CMessage& message)
{
                :
                :
  if((cmd == "FireShell") && (m_Status == Wait)){
  {
                :
                :
      ProjTraj_calculate(...);
                :
                :
  }
} return true;
```

< getMCResult 코드 >

```
                :
                :
advice execution("% CShell::ExtTransFn(...)") && that(shell) : after
(CShell& shell) {
    if ((shell.cmd == "FireShell") && (shell.m_Status == shell.Wait)) {
      ref_MonteResult->SetCPA(...);
    }
  }
```

[그림 13] 기법 적용 후 Shell 모델과 getMCResult

셋째, 모델 내부에서 바로 실험틀에 접근하는 관계가 DMMA에서 처리되어, 모델의 실험틀에 대한 의존성이 해소되었다. Shell 모델의 예를 다시 한 번 보자. 기법 적용 전인 그림 12에서는 Shell 모델 내부에서 실험틀의 클래스인 MonteResult에 바로 접근하지만, 기법 적용 후인 그림 13에서는 더 이상 Shell 모델 내부에서 MonteResult에 접근하지 않는다.

8. 효과 분석

제안한 적응성 향상 기법의 효과에 대한 이론적 분석은 4장에서, 실제 프로젝트에 적용한 결과에 대한 정성적 분석은 7장에서 제시하였다. 여기서는 효과를 정량적으로 분석하기 위해서 적응성에 관련된 세 가지 요구사항 변경 사례를 선정하고, 각 사례를 기존 구조(그림 8)와 새로운 구조(그림 9)에 적용했을 때 변경해야 할 요소(클래스, DMMA)의 개수를 조사하였다. 먼저 세 개의 구체적인 사례는 다음과 같다.

[표 1] 사례별 변경 요소 개수

사례	변경 요소 개수	
	기존 구조	새로운 구조
(1)	3	1
(2)	5	1
(3)	1	0

(1) Shell에서 수집하는 데이터 항목이 추가됨
(2) GunMotionCalc 모델이 모니터링 대상이 됨
(3) BCU 모델을 재사용하기 위해 분리시킴

세 개의 사례에 대해서 변경해야 할 요소의 개수는 표 1에 정리되어 있다.

사례 (1)과 (2)는 실험틀의 변경용이성, 즉 실험틀의 변경이 모델에 미치는 영향을 알아보려는 것이다. 그래서, 표 1에서는 두 사례가 실험틀 외부에 유발하는 변경에 대해서만 고려하였다. 사례 (3)은 모델의 재사용성에 관한 것이다. 재사용 대상 모델이 새로운 모델 구조와 호환가능하다고 가정할 때, 변경해야 할 요소가 어떤 것인지 조사하였다.

표 1을 살펴보면, 각 사례별로 변경해야 할 개수가 새로운 구조의 경우 상대적으로 작음을 알 수 있다. 기존 구조에서는 수집된 자료를 전송하기 위해 원자 모델 뿐 아니라 그것을 포함하는 결합 모델까지 변경해야 한다. 그러므로, 기존 구조의 경우 모델의 구조가 복잡할수록, 그리고 관련 모델의 개수가 많을수록 변경해야할 요소의 개수는 늘어난다. 하지만, 새로운 구조는 DMMA만이 변경대상이며 모델의 복잡도와 대상 모델의 개수와는 상관이 없다. 이것은 새로운 구조가 각 변경 사례에 대해 기존 구조보다 효율적으로 대응할 수 있다는 것을 뜻하고 결과적으로 기존 구조에 비해 적응성이 좋아졌다고 할 수 있다.

9. 결론 및 향후 과제

기존의 DEVS 기반 시뮬레이터는 구현 시 실험틀과 모델 사이의 상호 의존적인 구조로 인해, 실험틀을 변경할 때 모델의 변경이 유발되거나, 실험틀 관련 코드로 모델의 재사용성이 저해되었다. 이에 이

논문은 실험틀과 모델의 강한 의존성을 해소하여 실험틀의 변경용이성과 모델의 재사용성을 향상시키는 방안을 제안하였다. 기존 구조를 조사해서 문제점을 이해하고 DEVS 원자 모델의 분석을 통해 대상 모델의 특성을 파악하였다. 모니터링 관련 모듈의 분리가 기존 문제에 대한 해결책임을 인지하고 이를 가장 효과적으로 구현할 수 있는 기술로 AOP를 선정하였다. 실제 프로젝트를 통해 제안한 기술의 적응성 향상 효과에 대한 이론적, 정성적, 정량적 분석을 제시하였다.

향후 연구과제는 프로젝트에서 사용된 DMMA를 추상 애스펙트(abstract aspect)로 만들어 재사용성을 높이고 이를 더 발전시켜 DEVS 모델용 모니터링 프레임워크를 제작하는 연구이다. 또, 시뮬레이터 제어 관련 모듈을 실험틀과 분리시키는 것도 현재 고려하고 있는 연구이다.

참 고 문 헌

[1] B. Zeigler, H. Pracehofer and TagGon Kim, *Theory of Modeling and Simulation*, *Second Edition*, Academic Press, New York, 2000.

[2] Systems Modeling Simulation Lab., *DEVSimHLA v2.2.0 Developer's Manual*, Dept. of Electrical Engineering & Computer Science, KAIST, 2004.

[3] Tag Gon Kim, *Introduction to DEVS Formalism*, Systems Modeling Simulation Lab, Dept of Electrical Engineering and Computer Science, KAIST, 2001.

[4] Evans, Michael W. & Marciniak, John. Software Quality Assurance and Management. New York, NY : John Wiley & Sons, Inc., 1987.

[5] Lisa Wells, *Performance Analysis using Coloured Petri Nets*, 10th IEEE Int.l Symp. on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS 02), 2002.

[6] Mamadou K. Traoré, Alexandre Muzy, *Capturing the dual relationship between simulation models and their context*, Simulation Modelling Practice and Theory 14, 2006.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, *Aspect-Oriented Programming*, 11th European Conference on Object-Oriented Programming (ECOOP), 1997.

[8] Franco Cicirelli, Angelo Furfaro, Libero Nigro, Francesco Pupo, *MODULAR MODELLING AND ANALYSIS OF TIME-DEPENDENT SYSTEMS*, Proceedings 19th European Conference on Modelling and Simulation 2005.

[9] O. Dalle, *OSA : an Open Component-based Architecture for Discrete event Simulation*, in Proceedings of the 20th European Conference on Modeling and Simulation(ECMS2006). Bonn, Germany : ECMS & SCS, 2006.

[10] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. *AspectC++ : An aspect oriented extension to C++,* In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems(TOOLS Pacific '02), pages 53.60, Sydney, Australia, February 2002.

[11] Daniel Lohmann and Olaf Spinczyk, *On Typesafe Aspect Implementations in C++*, Proceedings of Software Composition(SC 2005), Edinburgh, UK, April, 2005.

[12] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, *An Overview of AspectJ*. In J. L. Knudsen, editor, ECOOP 2001 - Object-Oriented Programming, volume 2072 of LNCS. Springer-Verlag, June 2001.

[13] 이동훈, *함포중심 교전 시뮬레이션 시스템 개발*, 국방과학연구소, 2006.