임베디드 소프트웨어 설계 모델에서 비정상적 행위에 대한 상태전이 패턴의 적용

오기영○, 김상순, 홍장의
충북대학교 전자계산학과 소프트웨어공학연구실
ohgy@selab.cbnu.ac.kr○, sskim@selab.cbnu.ac.kr, jehong@chugnbuk.ac.kr

Applying a State Transition Pattern on Abnormal Behavior in Embedded Software Design Model

Gi-Young Oh○, Sang-Soon Kim, Jang-Eui Hong
Chungbuk National University Department of Computer Science

요 약

임베디드 소프트웨어 모델링에서 패턴의 활용은 설계 모델의 품질을 향상시키는데 매우 중요한 역할을 수행한다. 특히 상태 기반의 모델링은 임베디드 시스템의 행위를 중심으로 표현되기 때문에 패턴의 활용은 정확한 기능의 설계 및 설계 모델의 복잡도 감소에 도움이 된다. 본 연구에서는 임베디드 소프트웨어의 상태 전이 모델의 복잡도를 감소시키기 위해 제시된 기존의 설계 패턴을 고찰하고, 이에 대한 신택스와 시맨틱의 확장을 통해 새로운 설계 패턴을 제시한다. 제시된 설계 패턴은 임베디드 소프트웨어가 갖는 비예측성(unexpected) 이벤트를 반영할 수 있도록 확장되었으며, 이는 보다 유연하고, 확장 가능한 임베디드 소프트웨어의 모델링을 가능하도록 할 것이다.

1. 서론

최근 임베디드 소프트웨어의 적용 범위가 넓어지고, 응용 영역이 확장되면서 효과적이고 체계적인 임베디드 소프트웨어 개발 기술이 연구 제안되고 있다. 이러한 연구들은 임베디드 소프트웨어의 아키텍처 모델링을 비롯하여 제품 계열 공학, 컴포넌트 재사용 및 패턴, 가상 프로토타입 환경에서의 임베디드 소프트웨어 검증 등 매우 다양한 영역에서 이루어지고 있다[1, 2].

이중에서 컴포넌트 재사용 및 패턴에 대한 연구는 정형적이고 일반화된 문제에 대하여 체계적이고 최적화된 솔루션을 제공하기 위한 것으로써, 소프트웨어의 설계 과정에서 널리 활용되고 있다. E. Gamma[3], D. Alur[4], D. Schmidt[5] 등은 풍부한 경험과 지식을 기반으로 설계 패턴을 제안하였으며, 이들은 현재 많은 소프트웨어 개발 과정에서 활용되고 있다.

소프트웨어의 설계 과정에서 패턴의 활용은 주어진 문제에 대한 합리적인 설계 모델을 제공할 뿐만 아니라, 검증된 패턴의 활용으로 인한 모델의 적합성 및 신뢰성을 향상시킬 수 있다. 임베디드 소프트웨어 개발의 경우, 설계 모델은 외부 인터페이스의 다양성 및 하드웨어적인 특성을 고려하기 때문에 일반적인 트랜잭션 중심의 소프트웨어보다 더욱 복잡해진다. 따라서 기존에 제시되고 있는 다양한 임베디드 소프트웨어의 패턴을 활용하여 소프트웨어를 설계하는 경우 더욱 높은 품질의 소프트웨어 개발이 가능해진다.

임베디드 소프트웨어에 관한 설계 패턴에 대한 대표적인 연구는 M. Pont가 제안한 설계 패턴에 대한 연구[7]가 있다. 이 연구에서는 하드웨어 의존적인 설계 패턴을 72가지 제안하고 있으며, Scheduler, Driver 등에 대한 다양한 패턴을 제시하고 있다. 임베디드 소프트웨어에 대한 패턴 연구의 또 다른 하나는 Douglas에 의해 제안된 패턴 기반의 임베디드 소프트웨어 개발 방법이다. Harmony 프로세스라고 불리는 이 방법은 분석, 아키텍처 설계, 및 메카니스틱(mechanistic) 설계 과정의 전반에 걸쳐 패턴 기반의 임베디드 소프트웨어를 정의하고 있다.

Douglas에 의해 제시된 임베디드 소프트웨어 설계 패턴 중에서 상태 기반의 설계 패턴들[6]은 Latch State Pattern, Polling State Pattern, Exception State Pattern, Any State Pattern 등이 있는데, 이들 패턴은 임베디드 소프트웨어의 상세 설계 모델에 대한 행위를 표현하기 위해 제안되었다. Douglas에 의해 제시된 상태 기반의 패턴 중에서 Any State Pattern은 임베디드 시스템이 동일한 조건하에서 모든 상태로 전이가 가능한 시스템을 효율적으로 모델링하기 위해 제안되었다. 그러나 이 패턴의 특성은 패턴이 시스템의 기능적 또는 비기능적 특성을 표현하는 것이 아니라 단순히 설계 모델의 복잡도를 감소시키기 위한 목적으로 제안되었다는 것이다.

본 연구에서는 Douglas가 제안한 Any State Pattern에 대한 활용성을 높이기 위하여 제안된 패턴을 확장하였다. 임베디드 시스템이 갖는 상태 전이는 항상 주어진

조건과 예측된 상태 전이만이 존재하는 것이 아니라, 예측 불가능한 이벤트들에 대하여도 합당한 응답을 주어야 한다. 따라서 Any State Pattern을 활용하여 단순한 상태전이 이외에도 예외 처리를 위한 상태 전이가 가능하도록 하였다. 이렇게 함으로써, Any State Pattern의 적용 영역이 넓어지며, 또한 모델의 복잡도 감소로 인한 가독성 향상에 기여할 수 있을 것이다.

본 논문의 2장에서는 설계 패턴의 정의 형식에 대하여 살펴보고, 3장에서는 Any State Pattern에 대한 고찰을, 4장에서는 확장된 패턴에 대한 정의 및 적용에 대하여 기술하고, 5장에서는 예제 모델의 적용을, 그리고 6장에서는 결론을 기술하였다.

2. 설계 패턴의 정의 형식

소프트웨어의 설계 패턴이 활용되기 위해서는 무엇보다도 패턴에 대한 정확하고 풍부한 정보가 정의되어야 한다. 일반적으로 설계 패턴을 정의하기 위한 형식[3]은 다음과 같이 구성된다.

(1) Pattern Name: 패턴의 이름
(2) Intent: 패턴이 언급하고자 하는 이슈 또는 문제점
(3) Motivation: 문제를 설명하는 시나리오
(4) Applicability: 패턴의 적용 대상이나 적용 상황
(5) Structure: 패턴의 모양을 표현하는 다이어그램
(6) Participants: 패턴에 참여하는 객체 및 객체의 역할
(7) Collaborations: 패턴의 구성 객체간의 상호작용
(8) Consequence: 패턴의 활용에 대한 이득 및 장점
(9) Known Uses: 패턴이 적용된 예제

위와 같이 형식에 따라 패턴이 정의되며, 이중에서 Pattern Name, Intent, Pattern Structure 그리고 Consequence는 패턴의 정의에 있어서 필수적인 요소로 고려되고 있다.

3. Any State pattern

Douglas에 의해 제안된 Any State Pattern은 임베디드 시스템이 존재할 수 있는 특정한 상태들의 집합에서 모든 상태로의 상호 전이가 가능한 시스템을 모델링하기 위해 사용한다. 그림 1은 Any State Pattern을 적용하기 위한 논리적인 모델이다. 이 모델에서 모든 상태들은 상호 연결되어 있고(즉, 상태 전이가 가능하고), 각 상태 전이는 동등한 조건에서 이루어짐을 나타낸다.

그림 1에 나타난 상태 전이도는 모델의 가독성 측면에서 복잡도가 매우 높다. 즉 하나의 새로운 상태가 추가될 때마다 상태 전이의 수는 2배 증가하게 됨으로, n개의 상태에 대해서는 2^n개의 상태 전이를 갖게 된다. 이와 같은 모델의 복잡도 증가를 감소시키기 위해 Douglas가 제시한 Ant State Pattern은 그림 2와 같다.

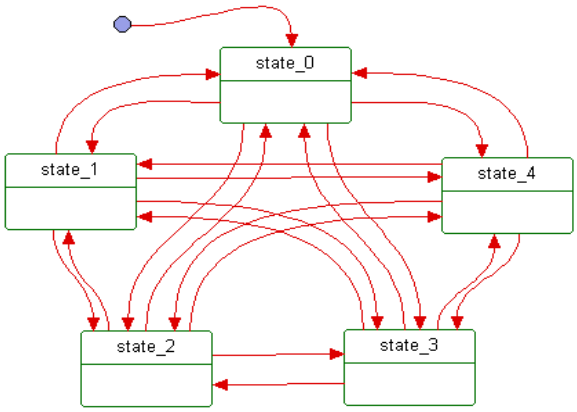

그림 1. 모든 상태로의 전이가 가능한 모델의 예

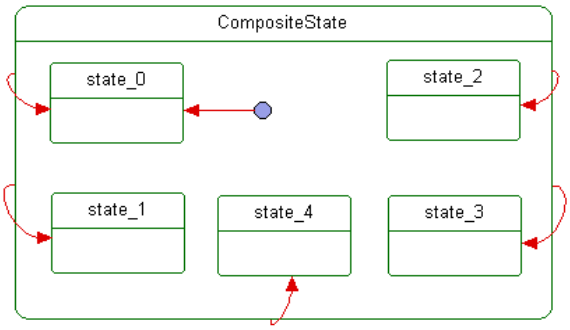

그림 2. “그림 1”에 대한 Any State Pattern의 적용

그림 2에 보는 바와 같이 Any State Pattern을 적용하기 위해서는 먼저 모든 상태들을 포괄하는 Composite State를 정의하고, 이의 하위 상태로 기존의 모든 상태들을 배치한다. 또한 하위의 각 상태간의 전이는 정의된 Composite State를 거쳐 발생하도록 정의한다. 이와 같은 표현에서 Composite State는 마치 모든 상태 전이의 흐름이 통과하는 공용의 버스와 같은 역할을 수행하게 된다.

위와 같이 Any State Pattern은 모델의 복잡성을 감소시켜 가독성을 증진시키는 역할을 수행하는데, 불행히도 이 패턴을 적용하기 위해서는 모든 상태로의 전이가 가능한 동종(homogeneous)의 상태 집합에 대해서만 적용 가능하다는 것이다. 그러나 임베디드 시스템의 모델링에 있어서는 예측 불가능한 외부의 이벤트에 대응하거나, 시스템의 모니터링을 위한 특정한 상태의 모니터링이 요구되는 경우가 많다. 이러한 상황을 앞서 설명한 Any State Pattern에 의해 표현하기 위해서는 이 패턴의 부분적인 확장이 필요하게 된다.

4. Any State Pattern의 확장

4.1 확장의 동기

간단한 레코드 테이프 플레이어(Record Tape Player)를 가정하자. 레코드 플레이어는 Stop, Forward start, Backward start, Fast forward, Fast backward, Pause, Recoding 등의 상태를 갖는다. 이와 같은 상태는 사람에 의한 버튼 조작에 의해 모든 상태로의 전이가 가능하다. 그러나 플레이어의 동작을 멈추기 위해서는 각 상태들이 궁극적으로 Stop 상태로의 전이를 통한 종료가 이루어져야 한다.

또한 비정상적이기는 하지만 플레이어의 특정 동작 상태에서(Stop 상태를 제외한) 강제로 레코드 테이프를 빼내는 인터럽트가 발생하는 경우에는 무조건 Stop 상태로의 전이가 발생되어야 한다,

발생 가능한 또 하나의 예외 상황은 전원이 Off되는 상태에서 대한 변화이다. 동작 중에 전원이 Off되는 이벤트가 발생하는 해당 동작이 Sleep되는 상태로 빠지게 되며, 전원이 On되면 다시 동작을 Resume하게 된다.

위와 같은 상태들은 Douglas에 의해 제시한 Any State Pattern에 의한 표현이 어려운 상황이며, 이를 체계적으로 반영하는 패턴의 확장이 요구된다.

4.2 시맨틱과 신택스 정의

4.1절에서 설명한 예외적인 상황에 대한 표현을 위하여 Douglas의 Any State Pattern은 확장되어야 한다. 패턴의 확장은 상태, 상태전이, 전이조건, 그리고 Indicator의 4가지 측면에서 이루어질 수 있는데, 이에 대한 사항은 다음과 같다.

(1) 상태 집합의 확장

Any State Pattern에 있어서 상태 집합에 대한 특별한 확장은 없으며 기존의 State Chart[8]에서 제공하는 기본적인 상태 표현 심볼과 시맨틱을 변경 없이 사용한다[9].

(2) 상태전이 집합의 확장

앞서 설명한 Sleep 상태는 모든 상태에서 전이가 가능하지만 Resume 될 때는 이전의 상태로만 전이가 가능해진다. 따라서 이러한 Sleep 상태에서의 전이는 이전의 상태 정보에 기반한 상태 전이가 이루어져야 한다. 일반적으로 이러한 전이를 위해 History State Indicator(HSI)를 사용하고 있는데, 이는 외부의 상태에서 상태전이가 발생할 때, 사용된다. 그러나 Any State Pattern에서의 적용은 동일한 Composite State 내에서 HSI를 사용할 수 있어야 한다.

심볼	시맨틱
→ⓗ	$\exists t_i \in T_{cs} \mid (\exists e / \forall S_{i-1} \rightarrow S_i) \wedge (\exists e / S_i \rightarrow \exists S_{i-1})$ for $\forall S_i\ S_{i-1} \in S_{cs}$

위의 시맨틱 정의에서 S_i는 임의의 상태를, S_{cs}는 Composite State에 포함된 상태들의 집합을, t는 임의의 상태전의를, T_{cs}는 Composite State내에서의 상태전이 집합을, 그리고 e는 이벤트를 의미한다.

(3) 전이 조건(guard condition)의 확장

Any State Pattern에 있어서 전이 조건에 대한 특별한 확장도 수행하지 않았다. 이는 기존의 State Chart에서 제공하는 기본적인 전이조건 기술언어를 따른다[9, 10].

(4) Indicator의 확장

Any State Pattern에서 정의하지 않은 인터럽트 또는 예외처리와 같은 상황을 표현하기 위하여 UBI(Unexpected Behavior Indicator)를 추가하였다. 이는 Composite State내에 존재하는 모든 상태에서 인터럽트 등의 예외처리 이벤트가 발생하면 무조건 UBI와 연결된 상태 전이를 수행한다.

심볼	시맨틱
①	for $\forall S_i \in S_{cs}$, $\exists e_{INT} \in E / S_i \rightarrow$ ①.

이상에서와 같이 동등한 조건에서 모든 상태로의 전이가 가능한 상태전이 패턴, 즉 Any State Pattern에 대하여 확장하였다. 확장 패턴의 적용에 대한 사례는 다음의 4.3 절에서 기술한다.

4.3 확장된 패턴 정의

4.2절에 기술한 Any State Pattern의 확장된 내용을 포함하는 패턴의 정의는 다음과 같다.

(1) Pattern Name : Conditional Any State Pattern

(2) Intent : 모든 상태로의 전이가 가능한 상태전이도는 모델의 표현상 매우 복잡해지는 단점이 있다. 따라서 이를 추상화하여 이해하기 쉽게 모델링할 필요가 있다. 또한 인터럽트나 예외처리의 경우는 모든 상태로의 전이가 발생하는 것이 아니라 특정 상태로만 전이가 이루어져야 한다. Ant State Pattern은 이러한 예외적인 상황에 대하여 표현력이 부족하다.

(3) Motivation : 모든 상태로의 전이가 가능한 시스템 행위가 존재할 때, 인터럽트와 같은 예외 상황이 발생할 수 있다. 인터럽트가 발생하면 특별한 상태로만 전이가 가능하다. 이와 같은 행위에서 특별한 상태라 함은 시스템에서 사전에 정의된 상태이거나 또는 이전의 상태일 수 있다.

(4) Applicability : 외부 시스템 또는 액터로부터의 이벤트에 의해 행위가 결정되는 임베디드 시스템의 경우에 적용된다.

(5) Structure : 그림 3과 같이 확장된 패턴의 클래스

다이어그램을 표현할 수 있다.

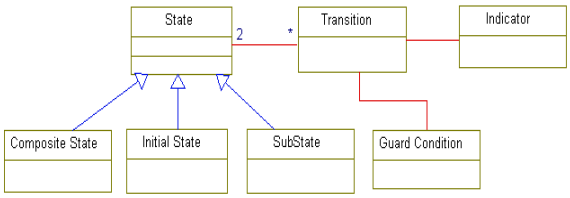

그림 3. 확장된 패턴의 클래스 다이어그램

(6) Participants: Composite State, Substates, Initial state, Transition, and Indicator
(7) Collaborations: State Chart에서 제공되는 행위적 특성을 반영한다.
(8) Consequence: 이 패턴의 활용에 대한 잇점은 인터럽트 등의 예외처리에 대한 행위적 특성을 Any State Pattern에 적용할 수 있으며, 이로 인하여 모델의 복잡도를 감소시킬 수 있다는 것이다.
(9) Known Uses: 가전제품의 리모트 컨트롤러, 레코드 테이프 플레이어, 그리고 제어 시스템에서의 테스트 기능 등이 확장된 패턴의 적용 가능한 예제들이다.

5. 확장 패턴 적용 사례

앞의 4.1에서 정의한 레코드 테이프 플레이어의 예를 이용하여 상태 전이도를 모델링하면 그림 4와 같으며, 이에 대하여 Conditional Any State Pattern을 적용한 상태전이도는 그림 5와 같다.

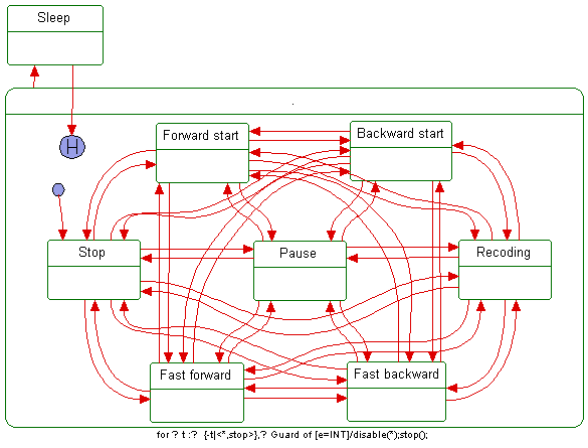

그림 4. 레코드 테이프 플레리어의 상태전이도

그림 4에서 보는 바와 같이 플레이어는 모든 상태에서 Stop 상태로 전이가 가능하지만, 인터럽트가 발생해도 동일하게 Stop 상태로 전이된다. 이와 같은 상황은 플레이어의 고유한 기능이라기보다는 예외적인 상황이기 때문에 별도로 분리하여 추가되었다. 또한 모든 상태에서 Sleep 상태로 전이가 가능하지만 Resume을 위해서는 바로 직전의 상태로 전이가 되어야 하다. 그림 5는 그림 4와 비교하여 상대적으로 모델의 복잡도가 감소했음을 알 수 있다.

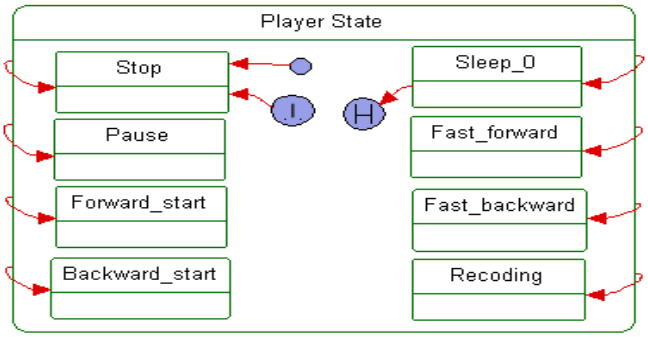

그림 5. 확장된 패턴이 적용된 플레이어의 상태전이도

6. 결론

본 연구에서는 임베디드 소프트웨어 설계를 위한 상태전이 패턴의 일종인 Any State Pattern에 대하여 살펴보고, 임베디드 시스템에서 발생 가능한 인터럽트의 개념을 반영한 Conditional Any State Pattern을 제시하였다. 제시한 패턴은 기본적인 Any State Pattern과 마찬가지로 복잡한 모델의 가독성을 향상시키기 위한 방법으로 활용될 수 있으며, 또한 기존의 패턴보다 더 융통성 있는 임베디드 시스템의 상태 전이를 표현할 수 있는 장점을 제공한다.

참고문헌

[1] Edward A. Lee, Embedded Software, Advance in Computer, Vol. 56, 2002
[2] N. Medvidovic, et al., "Software Architecture and Embedded Systems", IEEE Software, Vol. 22(5), Sep. 2005, pp. 883-86
[3] E. Gamma, et. al., Design Patterns, Addison-Wesley, 1995
[4] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns, Prentice-Hall, 2003
[5] D. Schmodt, et. al., Pattern-Oriented Software Architecture, Wiley, 2000
[6] B.P. Douglass, Real-Time UML Workshop for Embedded Systems, Newnes, 2007
[7] M. Pont and M. Banner, "Designing Embedded Systems Using Patterns: A Case Study, Journal of Systems and Software, Vol.71, 2004, pp.201-213
[8] D. Harel and M. Politi, Modeling Reactive Systems with Statecharts : The Statemate Approach, McGraw-Hill Companies, 1998
[9] H.-E. Eriksson, UML 2 ToolKit, Wiley, 2004
[10] B.P. Douglass, Real-Time UML, Third Edition, Addison-Wesley, 2004