임베디드 소프트웨어 행위 기술 모델링을 위한 시퀀스 다이어그램의 확장

이희진° 송인권 전상욱 홍장의*, 배두환
한국과학기술원 전자전산학과
*충북대학교 전기전자컴퓨터공학부
{hjlee°, igsong, sujeon, bae}@se.kaist.ac.kr, *jehong@chungbuk.ac.kr

Extended Sequence Diagrams For Embedded Software Behavior Modeling

Hee-Jin Lee °, In-Gwon Song, Sang-Uk Jeon, Jang-Eui Hong, Doo-Hwan Bae
Division of Computer Science, KAIST (Korea Advanced Institute of Science and Technology),
*School of Electronic and Computer Engineering, Chungbuk National University

요 약

임베디드 소프트웨어의 적용영역이 확장됨에 따라 학계와 업계에서 임베디드 소프트웨어 개발 기술에 대한 관심이 고조되고 있다. UML 2.0은 산업체에서 많이 사용되는 모델링 언어로, 그 동안 현장에서는 주로 상태머신 다이어그램(State Machine Diagrams)을 사용하여 임베디드 소프트웨어의 동적 행위를 모델링하여 왔다. 그러나 모델러는 상태머신 다이어그램보다 시퀀스 다이어그램(Sequence Diagrams)을 선호하는데, 이는 시퀀스 다이어그램을 사용하는 것이 직관적이고 정확한 행위 모델을 개발할 수 있기 때문이다. UML 2.0이 최근 다양한 모델링 요소를 반영하도록 확장되었음에도 불구하고, 시퀀스 다이어그램을 사용하여 임베디드 소프트웨어를 모델링을 하기에는 아직 부족한 점이 있다. 이를 보완하고, 임베디드 소프트웨어를 더욱 잘 디자인하기 위하여 예외 상황과 인터럽트를 처리할 수 있는 방법을 제안한다.

1. 서 론

임베디드 소프트웨어의 사용이 많아짐에 따라, 산업체와 학계에서 많은 관심이 주목되고 있다. 임베디드 소프트웨어가 일반적인 소프트웨어와 다른 점은 시간이나, 효율, 하드웨어 등의 분야에서 제약이 있다는 것이다. 이 제약을 보완하기 위하여, 체계적인 모델링이 필요하게 되었다.

UML(Unified Modeling Language)[1]은 산업체에서 가장 많이 사용되고 있는 모델링 언어이다. UML 다이어그램 중에서 임베디드 소프트웨어의 행위를 기술하기 위해 일반적으로 상태머신 다이어그램(StateMachine Diagrams)을 사용한다. 하지만 상태머신 다이어그램은 작성하거나 이해하기가 어려울 뿐 아니라, 객체들 간의 상호작용을 나타내주지 못하기 때문에 시스템을 이해하기가 쉽지 않다. 그에 반하여, 시퀀스 다이어그램(Sequence Diagrams)은 상태머신 다이어그램보다 사용하기 더 수월하다. 시퀀스 다이어그램은 직관적으로 다이어그램을 그릴 수 있고, 객체들 간의 상호작용이 모두 표현되기 때문이다.

UML 2.0 에서는 실시간 소프트웨어와 임베디드 소프트웨어를 중점에 두고 많은 개념들이 추가 되었다. 특히, MSCs(Message Sequence Charts)로부터 확장된 행위기술들이 UML 2.0 시퀀스 다이어그램에 추가 되었다[2, 3]. 본 논문에서는 임베디드 소프트웨어 모델링을 위해 추가해야 할 몇 가지 특성들을 제시하였다. 임베디드 소프트웨어의 특성인 인터럽트와 예외 상황 처리 개념을 시퀀스 다이어그램에 추가한다. 각 개념들을 사용하여 임베디드 소프트웨어를 시퀀스 다이어그램으로 잘 나타내도록 한다. 또한 예제를 통하여 그 쓰임새를 알아보도록 한다.

2. 시퀀스 다이어그램의 확장

시스템이 동작하면서 언제나 예외 상황과 인터럽트가 발생한다. 본 논문에서 예외 상황이란, 특정한 제약 조건이 부여된 행위에 대해서 이를 위반하였을 때 발생할 수 있는 상태를 말한다. 또한 인터럽트는 어느 행위를 수행하던 도중에 새로운 이벤트가 발생하고, 그 이벤트에 대한 처리를 먼저 수행하는 것을 말한다. 예외 상황과 인터럽트를 쉽고 간단하게 모델에 나타내는 것이 본 연구에서 제시하는 확장의 기본이다.

2.1. 예외상황 처리 Fragment

예외 상황은 임베디드 소프트웨어뿐만 아니라 많은 시스템에서도 빈번히 일어난다. 그럼에도 불구하고, UML 2.0에는 예외 상황을 처리하는 방법이 명확히 제시되지 않았기 때문에, 예외 상황을 처리할 수 있는 Fragment 'try'를 제안한다.

<표 1>에는 예외 상황을 처리하는 데 필요한 심벌들을 나타내고 있다. 예외 상황은 크게 예외상황의 발생과 그것을 처리하는 시나리오로 나누어 표현된다[4]. Fragment 'try'의 제일

위 칸에서는 예외 상황이 발생할 가능성이 있는 시나리오를 기술한다. 그 다음 칸들에서는 각 예외 상황에 맞는 처리 시나리오가 기술된다.

표 1. Interaction operator 'try'에 사용되는 심벌들

개념	심벌
Fragment 'try'	<<ExtendedOp>> try [isClass(insexc.Exc)]
Catch message	<<Catch>>catch(e1)

<표 1>에서 보는 바와 같이 isClass(insexc, Exc)라는 조건에 합당한 예외 상황이 들어왔을 때, 해당 칸의 처리과정이 실행되게 된다. 예외 상황의 종류는 네 가지로 나눌 수 있다[5, 6].

- 시간 간격 제한 예외상황(Duration Constraint Exception)
  일정하게 정해진 시간 간격 안에 시나리오가 완료되지 않았을 경우에 발생하는 예외 상황
- 시간제한 예외상황(Time Constraint Exception)
  정해진 시각에 시나리오가 처리 되지 않았을 때 발생하는 예외 상황
- 비 성공 시나리오 예외상황(Negative Scenario Exception)
  실행되어서는 안 되는 시나리오 (Fragment 'Neg')가 실행되었을 때 발생한다.

2.2. 인터럽트 처리 fragment

인터럽트는 시나리오가 진행되는 도중에 발생하는 다른 행위를 의미한다. 또한 기존의 시나리오보다 우선적으로 행해진다. 이것은 예외 상황과는 달리, 특정 이벤트에 종속적인 행동이 아닌 독립적으로 실행될 수 있는 행동이다.

UML 2.0 시퀀스 다이어그램에서 인터럽트의 발생을 나타낼 수 있지만, 발생을 예측할 수 없는 인터럽트는 나타내지 못한다. 지금까지는 하나의 인터럽트라도 발생 시기에 따라서 각각의 시퀀스 다이어그램에 별도로 나타내었다. 그러나 언제 발생할지 예측하기 힘든 인터럽트는 시퀀스 다이어그램에 표현하기 어려웠다.

시퀀스 다이어그램은 전체 시스템의 행위를 표현하기 보다는 부분적인 행위에 대한 시나리오를 기술한다. 즉 여러 개의 시나리오 중, 하나의 시나리오를 한 다이어그램에 표현하는 것이다. 그러나 언제 발생할지 예상할 수 없는 인터럽트를 표현하기 위해서는 시퀀스 다이어그램이 부분 시나리오가 아닌 전체를 나타내야 한다.

본 논문에서 제시하는 확장은 인터럽트가 언제 발생할지를 예상할 수 없는 상황에서 시퀀스 다이어그램에 인터럽트를 표시하는 방법이다. 그것은 언제 발생할지 모르는 하나의 인터럽트 때문에 너무나 많은 시퀀스 다이어그램을 그리는 것을 막기 위해서이다.

<표 2>에서는 본 연구에서 제시하는 인터럽트 처리에 관한 심벌들이 정의되어 있다. Fragment 'Interrupt'는 두 부분으로 나뉘어져 있다. 위 칸에서는 인터럽트가 발생할 가능성이 있는 기존의 시나리오가 나타난다. 다음 칸에는 인터럽트가 발생하는 경우, 해당 인터럽트를 처리하는 시나리오를 기술한다. 인터럽트 처리 시나리오는 인터럽트 메시지로부터 시작이 되는데, 이 메시지는 긴 점선으로 되어 있는 육각형 안에 표시된다. 그리고 모든 인터럽트 처리 시나리오를 마쳤을 때, 마지막에 'return' 메시지가 있으면 인터럽트가 발생한 순간으로 되돌아가서 다시 시작함을 나타낸다.

표 2. Interaction operator 'Interrupt'의 심벌들

개념	심벌
Fragment 'Interrupt'	<<ExtendedOP>> Interrupt
Interrupt signal	
Return message	return

인터럽트는 UML 2.0에서 제시된 Fragment 'alt'로도 나타낼 수 있지만, 인터럽트 발생 시점이 정확하지 않기 때문에 가능성이 있는 모든 발생가능 시점에 Fragment 'alt'를 사용해야 한다. 따라서 이러한 모델링은 간단한 시퀀스 다이어그램이 인터럽트 처리 시나리오 때문에 읽기 힘들 정도로 복잡해진다. 또한 실제 구현에 있어서 의도하지 않은 결과를 초래할 수 있다.

3. 예제 시스템 모델링

본 장에서는 앞에서 제안한 인터럽트와 예외의 개념을 적용한 예제를 제시한다. 여기서 사용되는 예제 시나리오에서는 "동영상 파일을 재생하기 위해 파일을 디코드 하는 도중 예외

상황이 일어나고, 그 이후 동영상 재생 중에 인터럽트가 발생" 하는 행위를 표현하였다.

<그림1>은 예외 상황을 다루는 시나리오를 나타내는 시퀀스 다이어그램이다. 그림 1에서 객체 MovieFileCodec에서 객체 FrameDecoder에게 동영상 파일을 디코드 하도록 메시지를 넘겨준다. 만약 객체 FrameDecode가 정해진 시간 안에 디코딩 행위를 끝나지 않는다면, 예외 상황이 발생하게 되어 동영상 파일을 재생하지 못한 채 끝나게 된다.

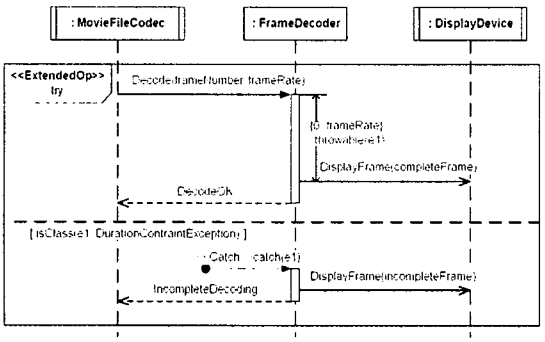

그림 1 예외상황 처리 예제 시나리오

<그림2>는 인터럽트를 다루는 시나리오를 나타내는 시퀀스 다이어그램 이다. 동영상 파일을 재생하는 도중에 fast forward 버튼이나 rewind 버튼이 눌려질 경우, 즉 객체 Button Controller가 객체 MoviePlayerController에게 pressed(fastforward), 또는 pressed(rewind) 메시지를 보내는 경우, 이를 인터럽트로 인식하고 해당 칸에 정의된 인터럽트 처리 부분(처리 시나리오)이 실행되게 된다. 각 인터럽트를 처리하는 시나리오가 끝난 이후의 return 메시지는 인터럽트가 발생하기 바로 직전의 상태로 돌아감을 의미한다.

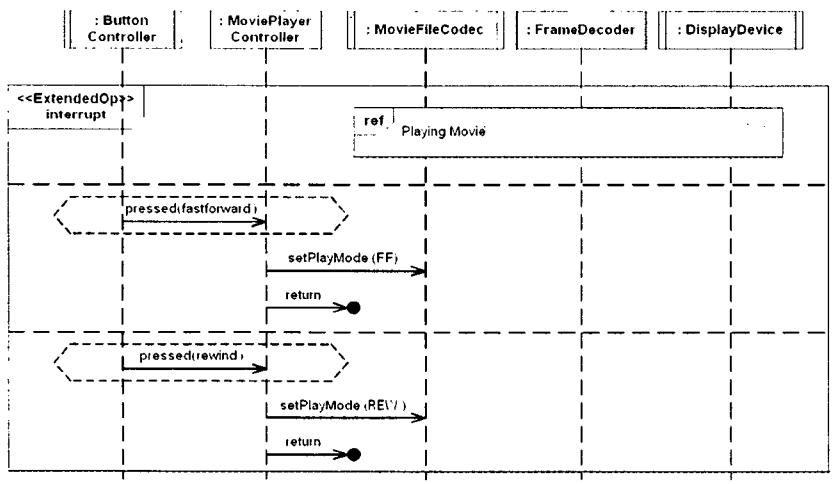

그림 2 Interrupt 처리 예제 시나리오

이와 같이 표현된 예외상황 처리와 인터럽트 처리에 대한 시퀀스 다이어그램 표현은 기존에 제공된 UML 2.0의 심벌만으로는 간단하게 표현하기 어려운 개념들이다. 따라서 제시하는 확장 개념을 이용한 임베디드 소프트웨어 모델링이 보다 간단하고 직관적으로 이루어질 수 있을 것이다.

4. 결론 및 향후 계획

본 논문에서는 UML 2.0 시퀀스 다이어그램에서 임베디드 소프트웨어를 모델링 하는데 부족한 점을 보완하여 확장 하였다. 논문에서 제시한 예외상황과 인터럽트의 표현법은 시퀀스 다이어그램이 나타낼 수 있는 시스템의 동적 행위에 대한 표현 범위를 넓혀 주었다. 임베디드 소프트웨어의 특징이라고 할 수 있는 예외상황과 인터럽트를 쉽게 표현할 수 있도록 함으로써, 업체의 소프트웨어 엔지니어들이 좀 더 직관적이고 정확한 모델링을 할 수 있게 하였다.

제시된 개념은 임베디드 소프트웨어의 모델링을 지원하는 CASE 도구에 반영되어 구현될 것이다. 현재 모델링 지원도구는 JDK 1.5를 이용하여 윈도우 XP상에서 개발 중에 있다.

참고문헌

[1] UML 2.0 Superstructure Specification, Available from : http://www.omg.org

[2] ITU,Z.120, in Message Sequence Chart(MSC),1999, ITU-T: Geneva. p.126.

[3] S. Mauw, M.A. Reniers, and T.A.C. Willemse. "Message Sequence Charts in the Software Engineering Process," CS-Reports 00-12, Department of Computing Science, Eindhoven University of Technology, 2000.

[4] H. Storrle, "Semantics of Control-Flow in UML 2.0 Activities", Visual Languages and Human Centric Computing, IEEE Symposium, Sept. 2004 pp. 235 - 242

[5] John B. Goodenough, "Structured exception handling", In Proceedings of the 2nd ACM SIGACTSIGPLAN symposium on Principles of programming languages, p.204-224, January 1975

[6] A. Strohmeier, S. Chachkov, "A side-by-side comparison of exception handling in Ada and Java", ACM SIGAda Ada Letters, Volumn XXI, Issue 3, Sep. 2001