XMI 기반 상태도의 소스코드 자동생성 엔진 구현☆

Code Generation from the Statechart Based on XMI

임 좌 상* 김 진 만**
Joasang, Lim Jinman, Kim

요 약

UML의 상태도는 CASE 도구마다 다르게 표현될 수 있어서 실제 적용에 어려움이 많지만, 시스템이 동작하는 측면을 효과적으로 나타낼 수 있다는 점에서 활용성이 높다. 상태도에서 소스를 생성하는 선행 연구를 보면, 단순한 분기문 또는 설계패턴 등을 적용하고 있지만 그 기술에 따라 생성된 소스코드에 차이가 있을 수 있다. 본 논문에서는 상태도를 문법형식에 맞게 엄격히 정의해서 소스코드를 생성하였다. 우선 서로 다른 CASE도구에서 작성된 상태도에서 추출된 XMI를 정제하였다. 여기서 EHA로 변환을 한 후 상태를 인식하였다. 그리고 상태도의 메타모델에 사용된 요소별로 자바 프로그래밍으로 전환하여 소스코드를 생성하고 기능성과 유지보수성을 측정하여 생성된 코드를 검증하였다. 본 논문에서 적용된 사례는 '에어컨'으로서, 다양한 상태와 전이가 포함되어 소스코드 생성에 적합하여 선정하였다. 그 결과 에어컨 사례의 상태도로부터 CASE 독립적인 XMI를 추출하여 정련한 후, 상태도에서의 동시성과 계층이 성공적으로 표현되었음을 확인했다. 향후 좀 더 큰 규모의 시스템에 적용하여 검증하는 연구가 필요하다.

ABSTRACT

Despite some practical confusion over the variations in the diagram which may be drawn differently depending upon the CASE, the statechart of UML has been widely used to show the dynamic behaviour of the systems. Prior research has employed either simple switch-case statement or the state design pattern to generate source code from the statechart, which may result in varying source codes. This paper made an attempt to formally define the statechart and generate source codes from it. Firstly we cleaned up the XMI which was generated from different CASEs. This XMI has been translated to the EHA to identify automata contained in it. Then the elements of the statechart metamodel were mapped to the java programs. We also verified the quality of source codes by measuring functionality and maintainability. The case employed in this study was the air conditioner. The reason was that the case includes various states and transitions of interest. It was found that XMI was well extracted by removing some legacy codes in the CASE and the source codes were then successfully generated with the concurrency and hierarchy of the statechart. Further research is required to validate it practical significance with a larger case.

☞ keyword : Statechart, XMI, Code Generation, 상태도, XMI, 소스코드생성

1. 서 론

소스코드 생성은 다양한 형태로 활용되고 있다. 예로서 우리가 익숙한 '컴파일러'는 고급언어를 기계코드로 생성하는 것이다. 또한 템플릿 등을 기반으로 소스코드를 생성하기도 한다. 본 논문에서는 설계모델에서 구현코드를 생성하는 것으로 한정한다. 이러한 연구는 1997년 OMG (Object Management Group)에서 만들어낸 시스템모델링 언어인 UML (Unified Modeling Language)이 실질적인 표준으로 업계에서 많이 사용되면서, 소스코드를 생성하는 것이 생산성, 오류를 줄이는 방안으로 환영받았기 때문이다. 2000년 이후에는 XML(eXtensible Markup Language)을 활용하여 모델 간 변환 및 코드 자동생성 연구[2,3]가 이루어졌다.

소프트웨어개발에 있어 설계모델은 개발자에게

* 정 회 원 : 상명대학교 디지털미디어학부 교수
jslim@smu.ac.kr (교신저자)
** 정 회 원 : 상명대학교 일반대학원 컴퓨터과학과 박사과정
hansumo81@gmail.com
[2011/05/09 투고 - 2011/05/13 심사(2011/08/16 2차) - 2011/09/05 심사완료]
☆ 본 연구는 2010학년도 상명대학교 교내연구비를 지원받아 수행되었습니다.

시스템 또는 제품의 추상적인 면을 보여줌으로 구현작업을 용이하게 만들고, 이에 기반한 소스코드 생성은 매뉴얼 코딩에서 발생되는 오류를 최소화하고 반복되는 코드의 수를 줄일 수 있어 소프트웨어 개발 생산성 향상을 가져온다[4].

UML의 메타모델은 다른 정형적인 언어로 구현이 가능한 구조로 되어있기 때문에 모델 표준화에 따라 소스코드 생성이 가능하다. 그러나 생성되는 코드의 양은 UML의 종류에 따라 차이가 난다. 예를 들어, 클래스다이어그램은 구조상 클래스명, 속성, 메소드, 클래스간 관계만을 나타내므로 이를 통해 생성된 소스코드의 양이 적다. 코드의 양을 늘리기 위해서는 시스템의 비즈니스 로직을 포함시켜야 한다. 이는 시스템의 동적 활동을 나타내는 것으로 UML의 상태도 또는 활동다이어그램을 통해 모델링 된다.

동적 모델을 이용하여 소스를 생성하는 연구는 주로 상태도를 대상으로 그 구성요소 (예: state, substate, composite state, transition)를 자바와 같은 정형적인 언어로 변경하는 생성기법을 제시했다[10]. 그러나 그 생성기법이 UML의 메타모델을 기반으로 하기보다는 생성된 상태도를 직접참조하고 있어, 개발자의 다이어그램 및 코드생성기법 이해 수준에 따라 생성되는 코드가 달라지는 문제점이 있다. 또한 상태도는 동시성, 계층과 같은 비정형적인 요소를 포함하고 있어, 표준화된 코드 생성에 어려움이 가중되고 있다.

본 연구에서는 XMI (XML Metadata Interchange)에 기반하여 상태도의 소스코드를 자동 생성하는 엔진을 구현하였다. 또한 EHA (Extended Hierarchical Automaton)를 활용하여 상태도에 포함된 비정형적인 요소를 표현하였다. 구현된 엔진의 코드 생성을 확인하기 위해 CASE (Computer Aided Software Engineering) 도구에서 생성한 상태도의 XMI 파일(상태도 메타모델)을 대상으로 소스코드를 생성했다. 그 결과 생성된 코드에는 동시성, 계층이 성공적으로 표현되었고, 상태도에 표현된 로직이 모두 구현되어 있음을 테스트케이스를 통하여 확인하였다.

(표 1) 상태도에서의 소스코드생성 연구

문헌	코드생성 방법	표현 구성요소
Ali[5]	State Pattern	이벤트, 액션, 상태 및 상태의 계층 표현
Niaz [6, 9, 11]	State Pattern	이벤트, 액션, 상태 표현
Pintér [7, 8]	EHA	이벤트, 액션, 상태, 가드 표현
Bokhari[17]	EHA	이벤트, 액션, 상태 표현
Blech[10]	Switch/ Case	이벤트, 액션, 상태 표현

2. 관련 연구와 접근 방법

2.1 상태도에서의 소스코드 생성 연구

UML의 다른 다이어그램에 비해 상태도는 시스템의 동적인 측면, 즉 객체나 시스템의 상태 표현에 매우 유용하다. 그럼에도 불구하고 상태도는 비정형적 요소로 인해 소스코드를 생성하는 것이 쉽지 않다[6,8]. 예를 들어, Pintér[8]는 UML 상태도가 계층, 동시성, 상호레벨전이 (interlevel transition)와 같은 상태 간 관계의 표현력이 충분하지 않음을 지적하였다. 이러한 문제는 '모델'과 '프로그램'과의 호환이 완전하지 않을 수 있음을 보여주고 있다.

상태도를 활용한 소스코드 생성 연구는 크게 (1) Switch/Case, (2) State Pattern, (3) EHA 기법을 사용하고 있다 (표 1 참조).

Switch/Case 기법[12]은 다른 기법에 비해 매우 간당한 방식으로 코드를 생성한다. 이는 상태도 모델링 도구인 Rhapsody[13]에서 적용되고 있지만 분기문의 사용으로 인해, 변경에 취약하고 동시성, 계층을 표현하는데 어려움이 따른다.

다음으로 State Pattern은 많은 연구에서 사용되고 있다[5,6,9,11]. 이 기법은 인터페이스의 사용으로 상태도의 모든 이벤트를 함수로 선언하고, 상태는 클래스로 인식하여 인터페이스를 상속받는 구조로 소스코드를 생성한다. 이는 상태 전이(transition)를 쉽게 코드로 인식하지만 Switch/Case 기법과 마찬가지로 동시성, 계층을 표현하는데 어

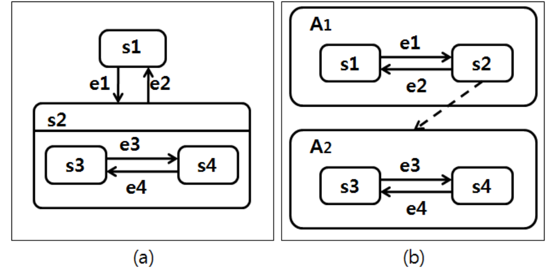
려움을 가지고 있다[8].

마지막으로, EHA는 상태도의 계층, 동시성, 전이를 정형화시키기 위한 기법으로 복합상태의 하위 상태를 자식 클래스로 인식하여 계층 관계를 코드화 할 수 있다. 이 외에 Pintér[8]의 연구에서 생성기법으로 AST (Action-State Tables)와 QHSM (The Quantum Hierarchical State Machine)을 소개하고 있다. AST는 상태와 관련된 이벤트를 테이블로 구현하여 한 이벤트가 특정 상태 내에서만 수행하도록 코드를 생성하는 것이다. 이는 Switch/Case 기법에 비해 빠르게 수행되지만 많은 메모리가 필요하고 마찬가지로 계층, 동시성을 지원하지 못하는 단점을 가지고 있다. QHSM은 이벤트 핸들러 방식에 기반하여 코드를 생성한다. 이는 계층과 전이의 효과적이고 유연한 구현을 지원하지만 전이와 연관된 액션을 직접적으로 표현할 수 없고, 동시성이 지원되지 않는다.

2.2 연구방법의 결정

2.2.1 XMI

본 연구에서는 소스코드 생성을 위해 XMI 모델을 적용한다. XMI는 OMG에서 2000년에 제시한 메타데이터 변환 표준이다. 현재 UML 모델은 CASE마다 표현 방법이 달라 호환성이 떨어지는 문제점이 있다. 하지만, 이러한 문제는 XMI의 사용으로 해결될 수 있다. 즉, CASE 간 모델교환이 XMI 사용으로 UML 모델의 정보 손실 없이 가능하게 된 것이다. 따라서 UML 모델은 XMI를 통해 플랫폼에 독립적인 메타데이터를 추출함으로 소스코드 생성이 가능하다. 이러한 이유로 XMI로부터 소스코드를 생성하는 연구가 최근 등장하고 있다. Senqupta [14]는 순차다이어그램을 OCL(Object Constraint Language)과 XMI를 사용하여 소스코드를 생성한 다음, 이를 입력으로 상태도를 생성하여 모델 간 호환성을 유지하였다. 또한 Sturm[15]는 XMI를 사용하여 소스코드를 직접적으로 생성하였다. 이 연구에서는 UML 모델로부터 XMI 파일을 생성하고, 그로 인해 개발품질이 향상될 수 있음을 주장하였다. 따라서 본 연구에서는 XMI를 적용하여 상태도 설계모델에서 소스코드 생성에 필요한 메타정보를 추출한다.



(그림 1) UML로 표현한 상태도 (a)와 EHA적용한 상태도 (b)의 비교

2.2.2 EHA

1장에서 상태도에서 소스코드를 생성하는 동안에 모델에서 표현된 정보가 손실되는 문제점을 지적한 바 있다. 본 연구에서는 이런 문제 가운데 계층과 동시성을 EHA를 적용하여 보완하고자 한다. EHA는 Mikk[16]의 연구에서 제안된 것으로, 상태도의 코드변경 시 표현하기 어려운 상호레벨전이, 동시성, 계층과 같은 관계를 오토마타 (Automata) 이론을 적용하여 해결하고 있다. 예를 들어, 상태도 전이의 이벤트는 (F, E, γ)와 같이 표현될 수 있다 (참조: [16] EHA 정의 3). 첫 번째 요소인 F 는 ‘순차적 오토마타’의 집합을, 두 번째 E 는 ‘이벤트 집합’을, 마지막 γ 은 F 에서의 ‘합성기능’을 나타낸다. 그림 1은 기존의 상태도 (a)를 변환하여 EHA (b)로 나타낸 것이다. 그림에서 보듯이, (a)의 상태 s1과 복합상태 s2가 (b)에서 오토마타 A_1으로 표현되었고 s1과 s2간의 이벤트 (e1, e2)를 담고 있다. 또한 (a)의 복합상태에 있는 상태 s3과 s4는 (b)의 오토마타 A_2로 표현되었고 A_1과 마찬가지로 이벤트를 담고 있다. 이처럼 상태도를 EHA기법으로 정제하면 복잡한 관계 (동시성, 상호레벨전이, 계층)가 단순하게 순차적으로 표현된다. (그림 1)의 (b)를 EHA 식으로 표현하면 식 (1)과 같다 (식에서의 γ 는 (b)에서의 화살표점선을 나타냄).

$$(\{A_1, A_2\}, \{e_1, e_2, e_3, e_4\}, \gamma) \quad 식(1)$$

이와 같이 EHA는 상태도의 복잡한 관계를 단순하게 나열하여 그 관계를 정해진 정의에 의해 조망해 볼 수 있다. 따라서 본 연구에서는 상태도의 복잡한 관계 정보를 추출하기 위해 이를 단순하게 나열할 수 있는 EHA 기법을 사용한다.

3. 자동 소스코드생성 알고리듬 구현

본 연구에서는 XMI로 표현된 상태도로부터 소스코드를 3단계에 걸쳐 자동 생성한다 (그림 2 참조: (1) EHA 변환기 (오토마타 추출), (2) 요소인식기 (메타정보의 요소관계 인식) 및 (3) 코드 매핑기 (소스코드 생성)). 단계별로 수행하는 작업을 상세하게 입력, 출력, 알고리듬으로 구분하여 설명하면 다음과 같다.

1단계 - EHA 변환기

XMI 메타정보로부터 오토마타를 추출한다.

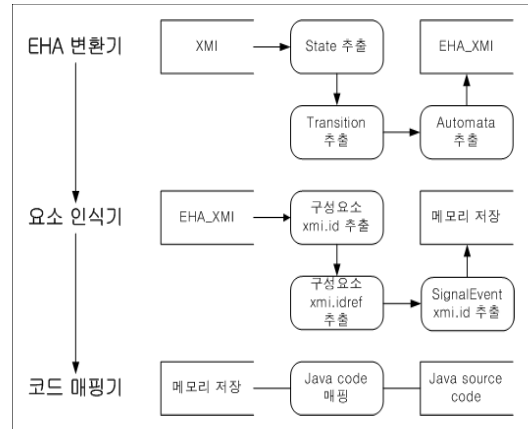
- 입력: 상태도의 XMI (CASE에서 자동 생성)
- 출력: XMI에서 인식된 오토마타 (그림 2)의 EHA_XMI)
- 알고리듬: XMI 파일에서 Unisys 코드*나 CASE도구 확장 정보**를 제외한다. 이후 그림 3의 (1), (2)를 통해 복합상태는 한 개의 오타마타로, 분기동기화 (fork)의 경우에는 전이의 수 만큼 오토마타로 인식한다.

2단계 - 요소인식기

XMI 메타정보에서 소스코드와 관계된 요소들을 인식한다.

- 입력: 오토마타 파일 (1단계 출력 EHA_XMI)
- 출력: 상태도의 각 구성요소 id (xmi.id, xmi.idref)와 전이 관계 정보

* XMI에 포함된 위치, 색 정보를 제외한다.
** Together, Rose 도구의 확장정보를 제외한다.



(그림 2) 본 연구에서 적용한 소스코드 생성절차

$$
\begin{array}{l}
(1)\ IF\ (current\ state == CompositeState) \\
\quad automata = current\ state \\
\\
(2)\ IF\ (PseudoState.kind == fork) \\
\quad FOR\ (PseudoState.transition.count) \\
\quad\quad IF\ (PseudoState.transition.idref == \\
\quad\quad\quad current\ state.id) \\
\quad\quad\quad sequential\ automata = current\ state \\
\\
(3)\ IF\ (Transition\ xmi.id == current\ state\ xmi.idref) \\
\quad current\ state = Transition.source \\
\quad action\ state = Transition.target \\
\quad current\ method\ id = Event\ xmi.idref \\
\\
(4)\ IF\ (SignalEvent\ xmi.id == current\ method\ id) \\
\quad current\ method\ name = SignalEvent.name
\end{array}
$$

(그림 3) XMI기반 소스코드 생성 알고리듬

- 알고리듬: EHA_XMI 파일에서 (그림 3)의 (3)을 사용하여 한 상태가 참조하고 있는 전이를 식별하고 이벤트가 발생 할 때 변환 되는 상태 정보를 추출하고 상태 전이를 발생시키는 이벤트 정보를 추출한다. 이 후 그림 3의 (4)를 사용하여 이벤트의 이름 정보를 추출한다.

3단계 - 코드매핑기

앞 2단계에서 출력된 상태도의 구성요소 정보 (상태, 전이, 이벤트)를 자바 구조체와 매핑하여 코드를 생성한다.

- 입력: 인식된 상태도 구성요소의 정보 (2단계

(표 2) 상태도 메타모델요소와 자바 구조체 매핑

상태도 메타모델요소	자바 구조체	비고
CompositeState, SimpleState	클래스	상속 (슈퍼, 서브 클래스)
Event	함수	실제 클래스 함수, 추상클래스 멤버함수
Guard	조건문	멤버함수 내 조건문

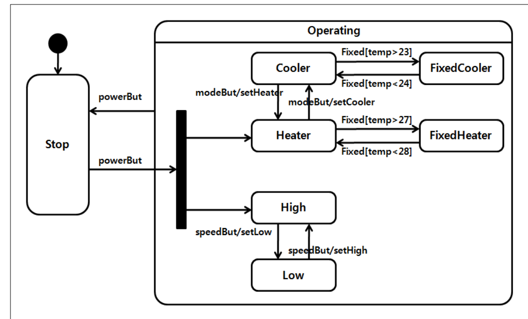
출력, 메모리에 저장)
- 출력: 자바 소스코드
- 알고리듬: (표 2)와 같이 상태도 메타모델요소를 자바 구조체와 매핑하여 실제 코드 정보(클래스 명, 메소드 명, 조건문)로 인식한다. SimpleState, CompositeState 요소는 각각 서브클래스와 슈퍼클래스로, Event와 Guard 요소는 메소드명과 메소드 내 로직 정보로 인식된다.

소스코드생성엔진은 XML처리가 쉬운 스크립트 언어인 Tcl로 구현되었다. UML은 현재 2.3이 발표되었지만 현재 많이 쓰이고 있는 UML 1.4를 대상으로 한다. CASE 도구간의 호환을 위해 XMI 1.1을 사용하였다. 본 연구에서는 CASE 도구로 Borland Together Architect 2005, IBM Rational Rose 7.0을 사용했다.

4. 적 용

4.1 사 례

본 연구에서는 Niaz[6]의 연구에서 사용한 사례를 선정하였다. 사례를 간단히 설명하면, (그림 5)에서 보듯이 사례는 크게 Stop 및 Operating 2개의 상태로 구성되어 있다. 첫 번째는 Stop상태로, 에어컨의 정지 상태를 말한다. 이 때 powerBut 이벤트가 발생하면 시스템이 구동되면서 두 번째 상태인 Operating상태로 전환된다. Operating 상태는 복합상태로 그 안에 6개의 하위상태를 포함하고 있다. 예로, Heater 상태는 modeBut 이벤트가 발생하면



(그림 5) 에어컨 시스템의 상태도

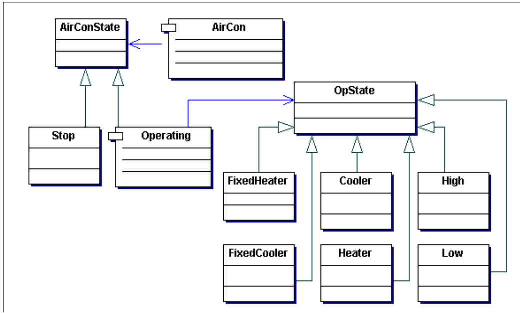
(표 3) 에어컨 시스템의 이벤트에 따른 상태 전환

원 상태	이벤트	전환 상태
Stop	powerBut	Operating
Operating	powerBut	Stop
Heater	modeBut	Cooler
	Fixed(temp>27)	FixedHeater
Cooler	modeBut	Heater
	Fixed(temp>23)	FixedCooler
High	speedBut	Low
Low	speedBut	High
FixedHeater	Fixed(temp<28)	Heater
FixedCooler	Fixed(temp<24)	Cooler

Cooler 상태로 Fixed(temp>27) 이벤트가 발생하면 FixedHeater 상태로 전환된다. 나머지 상태의 전환 관계는 (표 3)에 나타냈다.

4.2 소스코드 자동 생성

자동소스코드 생성엔진을 통해 생성된 소스코드는 (그림 6)과 같이 11개의 클래스로 이루어졌으며, 이 중 상황을 나타내는 클래스의 코드 2개 (시스템 전체를 나타내는 AirCon, 작동상태를 나타내는 Operating)를 (그림 7)에 나타냈다. 추상클래스는 실제 클래스의 공통 인터페이스를 추출하여 2개가 생성되었다 (AirConState, OpState). 상황클래스로부터 전달된 요청을 처리하는 실제클래스는 8개가 생성되었다 (AirConState 클래스를 상속받는 Stop, Operating



(그림 6) 에어컨 사례의 클래스 다이어그램

```
public class AirCon {
  public AirConState state;
  public AirConState stopState;
  public AirConState operatingState;
  public AirCon() {
    stopState = new Stop(this);
    operatingState = new Operating(this);
    state = stopState;   // default
  }
  public void setState(AirConState state) {
    this.state = state;
  }
  public void powerBut() {
    state.powerBut();
  }
}
```

(a) AirCon Class

```
public class Operating extends AirConState {
  OpState substate_f1; OpState substate_f2;
  public OpState coolerState;
  public OpState heaterState;
  ...
  int temp;
  public Operating(AirCon airCon) {
    coolerState = new Cooler(this);
    heaterState = new Heater(this);
    ...
    temp = 20;
  }
  public void setSubState(OpState substate1,
                          OpState substate2) {
    this.substate_f1 = substate1;
    this.substate_f2 = substate2;
  }
  public void powerBut() {
    ac.setState(ac.stopState);
  }
  public void modeBut() { ... }
  public void speedBut() { ... }
  public void Fixed() { ... }
}
```

(b) Operating Class

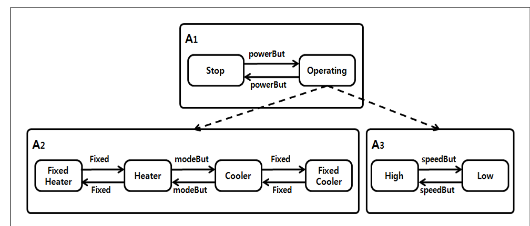
(그림 7) 에어컨 사례의 자동 생성된 소스코드
(전체 11개 클래스 중 일부)

```
<XMI.content>
 <UML:Model xmi.id = 'S.1' name = 'Project' visibility = 'public'>
  <UML:Namespace.ownedElement>
   <UML:StateMachine xmi.id = 'S.2'
         name = 'State/Activity Model' visibility = 'public' isSpecific
    <UML:StateMachine.top>
     <UML:CompositeState xmi.id = 'XX.39'
          name = '{top}' visibility = 'public' isSpecification = 'fals
      <UML:CompositeState.subvertex>
       <UML:Pseudostate xmi.id = 'G.1'
             name = 'StartState1' visibility = 'package' isSpecificatio
        <UML:StateVertex.outgoing>
         <UML:Transition xmi.idref = 'G.0'/>
        </UML:StateVertex.outgoing>
       </UML:Pseudostate>
       <UML:SimpleState xmi.id = 'G.3'
             name = 'Stop' visibility = 'package' isSpecification = 'fa
        <UML:StateVertex.outgoing>
         <UML:Transition xmi.idref = 'G.2'/>
        </UML:StateVertex.outgoing>
       </UML:SimpleState>
 ...
```

(그림 8) 에어컨 사례의 상태도 XMI (일부)

```
START {name StartState1} {xmi.id G.3} {xmi.idref G.2}
SState {name Stop} {xmi.id G.1} {xmi.idref G.0}
CompositeState {name Operating} {xmi.id G.27} {xmi.idref G.7}
InternalTransition {name entry} {xmi.id G.4} {source G.27} {target G.27
InternalAction {name entry} {name setOn}
InternalTransition {name exit} {xmi.id G.5} {source G.27} {target G.27}
...
CompositeSub {name Operating} {name Cooler} {xmi.id G.10} {xmi.idref G.
CompositeSub {name Operating} {name Heater} {xmi.id G.13} {xmi.idref G.
CompositeSub {name Operating}  {name High} {xmi.id G.15}
CompositeSub {name Operating} {name Low} {xmi.id G.17}
...
Transition {name {link: Stop -> History1}} {xmi.id G.0} {source G.1} {t
Transition {name {link: StartState1 -> Stop}} {xmi.id G.2} {source G.3}
Transition {name {link: Operating -> Stop}} {xmi.id G.7} {source G.27}
Transition {name {link: Cooler -> Heater}} {xmi.id G.8} {source G.10} {
TranAction {name setHeater} {xmi.idref G.30}
Transition {name {link: Cooler -> FixedCooler}} {xmi.id G.9} {source G.
TranGuard {body {temp < 24}} {xmi.idref G.31}
Transition {name {link: Heater -> Cooler}} {xmi.id G.11} {source G.13}
...
```

(그림 9) 에어컨 사례의 EHA (일부)



(그림 10) 에어컨 사례의 EHA

(AirCon 상황클래스 입장에서는 실제클래스로 인식 됨 (그림 6 참조)) OpState 클래스를 상속받는 Cooler, Heater, FixedCooler, FixedHeater, High, Low). 상태도의 가드는 (예: Cooler 상태일 때는 temp > 23) 상태 변환을 수행하는 함수 (예: Cooler 클래스의 Fixed()) 내의 조건문으로 생성하였다.

4.3 사례의 소스 생성을 위한 EHA 변환 과정

본 연구에서는 상태도의 동시성 및 계층관계를 나타내기 위해 XMI 정보가 EHA로 변환된다. 사례의 XMI는 (그림 8)과 같다. 이는 사례의 상태도를 CASE 도구를 통해 XMI 파일로 변환한 것이다. 본 논문에서는 현재 널리 사용되고 있는 대표적인 CASE (Together, Rose)를 사용하였다 (Together에서는 813, Rose는 1577 줄의 XMI). 두 개의 CASE를 사용한 이유는 생성된 XMI가 서로 차이가 있더라도 처리가 가능한지 확인하기 위해서였다.

(그림 9)는 XMI 정보로부터 EHA 기법을 적용하여 나온 결과이다. 보는바와 같이 상태와 전이의 정보들로 구성되어 있다.

사례의 EHA 변환을 도식화 하면 (그림 10)과 같다. 이러한 정보의 추출은 다음과 같은 EHA 변환식 ((2)~(4))을 통한 것이다.

$$\gamma = \{Operating \rightarrow \{A_2, A_3\}, Stop\}$$ 식(2)

$$\cup \left\{ s \rightarrow \varnothing \mid s \in \begin{Bmatrix} Heater, Cooler, FixedHeater, \\ FixedCooler, High, Low \end{Bmatrix} \right\}$$ 식(3)

사례 상태도의 상태 계층은 식 (2)와 같이 합성 기능 γ 로 변환하여 그 관계를 재설정 할 수 있다 (참조: [16] EHA 정의 2). (그림 10)에서와 같이 Operating 상태는 2개의 오토마타 (A_2, A_3)나 Stop으로 전환될 수 있다. 따라서 Operating을 2개 오토마타의 상위 상태로 인식할 수 있으며, 또한 동시성을 가진 상태임을 확인할 수 있다. 식 (3)은 Operating의 하위 상태 집합을 정의한 것으로 실제 클래스들의 후보가 될 수 있다.

$$HA = \left(\{A_1, A_2, A_3\}, \begin{Bmatrix} powerBut, modeBut, \\ speedBut, Fixed \end{Bmatrix}, \gamma \right)$$ 식(4)

식 (4)는 '계층적 오토마타'를 구하는 식으로 이를 통해 클래스의 오퍼레이션 후보를 추출 하게 된다 (참조: [16] EHA 정의 3).

이처럼 XMI 정보로부터 EHA 기법을 적용하여 동시성과 계층관계를 추출하여 소스코드를 생성하게 된다.

5. 평　가

본 연구에서는 (1) 사례의 상태도에 표현된 알고리듬이 모두 구현되었는지 테스트케이스를 통하여 기능성을 평가한다. 또한 (2) 보통 자동 생성된 소스코드는 이해가 어려운 점이 있으므로 유지보수성을 평가한다.

5.1 기능성

사례에서 사용된 가드는 (그림 5)에 나타난 바와 같이 총 4개이다 (예: Fixed[temp>23]). 따라서 (표 3)과 같은 테스트케이스를 인식하였고, 조건에 따라 모두 예상된 상태로의 변환을 확인하였다. 테스트케이스는 JUnit으로 프로그래밍 하였다.

(표 3) 사례에 대한 테스트 케이스 (Fixed() 메소드)

메소드	조건	예상 상태	결과 상태
Cooler.Fixed()	temp>23	FixedCooler	FixedCooler
Heater.Fixed()	temp>27	Cooler	Cooler
FixedCooler.Fixed()	temp<24	FixedHeater	FixedHeater
FixedHeater.Fixed()	temp<28	Heater	Heater

(표 4) 사례에 대한 n-switch 테스트 케이스

상태의 흐름	예상상태	결과상태
Stop->Operating	Operating	Operating
Operating->Stop	Stop	Stop
Stop->Operating->Stop	Stop	Stop
Operating->Stop->Operating	Operating	Operating
Cooler->Heater	Heater	Heater
Heater->Cooler	Cooler	Cooler
Cooler->Heater ->Cooler	Cooler	Cooler
Heater->Cooler->Heater	Heater	Heater
Low->High	High	High
High->Low	Low	Low
Low->High->Low	Low	Low
High->Low->High	High	High
Cooler->FixedCooler	FixedCooler	FixedCooler
FixedCooler->Cooler	Cooler	Cooler
Cooler->FixedCooler->Cooler	Cooler	Cooler
FixedCooler->Cooler->FixedCooler	FixedCooler	FixedCooler
Heater->FixedHeater	FixedHeater	FixedHeater
FixedHeater->Heater	Heater	Heater
Heater->FixedHeater->Heater	Heater	Heater
FixedHeater->Heater->FixedHeater	FixedHeater	FixedHeater

다음으로 상태의 변환과정의 테스트는 n-switch 방식으로 실행하였다[18]. 사례에서 가능한 8개의 상태 (Stop, Operating, Heater, Cooler, High, Low, FixedCooler, FixedHeater)로부터 테스트케이스를 인식하여 JUnit으로 실행한 결과 모두 예상된 상태로 변환되는 것으로 나타났다 (표 4 참조).

5.2 유지보수성

자동 생성된 소스코드는 저자가 직접 작성한 매뉴얼코딩과 주관적으로 비교하였다. 자동 생성된 코드는 매뉴얼코딩과 유사하게 상태패턴에 따라 프로그램된 것으로, 상황에 따라 실행하는 클래스로 모듈화 되어있어 이해성이 좋은 편이었다. 또한 생성된 소스코드의 유지보수성을 Coleman[19]의 MI (Maintainability Index)지표를 적용하여 계량적으로 평가하였다. MI식에는 소스코드를 단어분량으로 (LOC가 아닌) 계산한 HV (Halstead Volume)과 분기정도에 따라 복잡성을 나타내는 CC (Cyclomatic Complexity)를 사용하고 있다 (식 (5) 참조). 본 사례에서 생성한 소스코드의 MI는 36.3으로 계산되었다 (HV=581, CC=6, LOC=131)이었다. 이 결과는 유지보수성이 양호한 (green rating 20~100) 범위에 해당된다.

$$MI = MAX(0, (171 - 5.2 \times \ln(HV) - 0.23 \times (CC) - 16.2 \times \ln(LOC)) \times 100/171)$$

식(5)

6. 결론 및 향후 연구 방향

소프트웨어나 시스템의 설계에 있어 UML이 실제적인 표준으로 널리 적용되면서 소스코드생성 연구가 많이 등장하고 있다. 특히 상태도는 다른 다이어그램에서 표현하기 힘든 동적인 면을 나타낼 수 있다는 점에서 그 활용성이 높다. 반면 상태도의 메타모델 요소를 코드로 생성하면서 적용된 기법에 따라 (단순 분기문 또는 설계패턴), 생성된 프로그램이 차이를 보여 실무에 적용하기에 어려움이 있다. 특히 XMI가 적용하는 CASE마다 생성되는 코드에 편차가 있고, 상태도의 메타모델요소의 정형적 정의가 부족하다.

이 점에 착안하여 본 연구에서는 실무에서 많이 사용하는 2개의 CASE 도구를 적용하여 XMI를 생성하였다. 여기서 생성된 코드는 서로 차이를 보여서 정련한 후 소스코드 생성에 활용하였다. 정련된 XMI는 EHA에 따라 정형적으로 정의하여, 상태도의 메타모델요소를 자바프로그램으로 변환하였다. 그 결과 CASE에서 각각에서 모델링 한 에어컨 시스템 사례로부터 동일한 자바소스코드가 생성되었으며 코드에는 동시성, 계층관계가 표현되었다. 또한 본 연구에서 구현한 알고리듬의 타당성을 (1) 가드와 상태에 대해 테스트케이스를 인식하여 모델변환과정에서의 타당성을 검증하였고, 더불어 (2) 유지보수지표를 통하여 평가하였다.

요약하면, 본 논문은 (1) 모델변환 표준인 XMI를 사용하여 CASE에 독립적으로 소스코드를 생성할 수 있고, 매뉴얼코딩에서 발생할 수 있는 문제점을 회피할 수 있으며, (2) EHA 기법의 사용으로 생성된 코드에 상태 간 동시성과 계층관계가 코드로 표현되어 기존 연구에서의 어려움을 해결하였다. 그러나 본 연구에 적용된 사례는 그 규모가 크지 않아, 타당성을 검증하기 위해서는 실제 사례를 대상으로 후속 연구가 이루어져야 한다.

참 고 문 헌

[1] Czarnecki, K. and U. Eisenecker, ‘Generative Programming: Methods, Tools, and Applications’, Addison-Wesley, 2000.

[2] Zhang, H., S. Jarzabek, and S. Soe Myat, ‘XVCL approach to separating concerns in product family assets’, Erfurt, Germany: Springer-Verlag, 2001.

[3] Cleaveland, J.C., ‘Program Generators with XML and Java’, Prentice Hall, 2001.

[4] Ray, W.J. and A. Farrar, ‘Object Model Driven Code Generation for the Enterprise, in Rapid System Prototyping’, IEEE International Workshop. 2001.

[5] Ali, J. and J. Tanaka, ‘Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams’, ACIS Int. J Comp. Inf. Sci., VOL.2(1), pp.24-36, 2001.

[6] Niaz, I.A. and J. Tanaka, 'CODE GENERATION FROM UML STATECHARTS', The 7th IASTED International Conference on Intelligent Systems and Control (SEA2003), 2003.

[7] Pintér, G. and I. Majzik, 'AUTOMATIC CODE GENERATION BASED ON FORMALLY ANALYZED UML STATECHART MODELS', Formal Methods for Railway Operation and Control Systems, 2003.

[8] Pintér, G. and I. Majzik, 'PROGRAM CODE GENERATION BASED ON UML STATECHART MODELS', Periodica Polytechnica Electrical Engineering, VOL. 47, NO. 3–4, pp.187–204, 2003.

[9] Niaz, I.A. and J. Tanaka, 'MAPPING UML STATECHARTS TO JAVA CODE', IASTED International Conf. on Software Engineering (SE2004), 2004.

[10] Blech, J.O., S. Glesner, and J. Leitner, 'Formal Verification of Java Code Generation from UML Models', the 3rd International Fujaba Days 2005, 2005.

[11] Niaz, I.A. and J. Tanaka, 'An Object-Oriented Approach To Generate Java Code From UML Statecharts', Computer & Information Science, VOL. 6, NO. 2, 2005.

[12] Douglass, B.P. and D. Harel, 'Real-Time UML: Developing Efficient Objects for Embedded Systems', Addison Wesley Longman, 1998.

[13] Telelogic Inc., Rhapsody, http://modeling.telelogic.com/

[14] Sengupta, S., A. Kanjilal, and S. Bhattacharya, 'Automated Translation of behavioral models using OCL and XML', TENCON 2005 IEEE Region 10, pp.1-6, 2005.

[15] Sturm, T., J.v. Voss, and M. Boger, 'Generating Code from UML with Velocity Templates', the 5th International Conference on The Unified Modeling Language, Springer-Verlag, 2002.

[16] Mikk, E., Y. Lakhnechi, and M. Siegel. 'Hierarchical automata as model for statecharts', In R. Shyamasundar and K. Euda, editors, Third Asian Computing Science Conference. Advances in Computing Sience – ASIAN'97, volume 1345 of Lecture Notes in Computer Science, Springer-Verlag, pp.181-196, 1997.

[17] Bokhari, A. and S. Poehlman, 'Formalization of UML State-Charts: Approaches for Handling Composite States', Department of Computing & Software, McMaster University, Technical Report CAS, 2005.

[18] Chow, T. S., 'Testing Software Design Modeled by Finite-State Machines', Software Engineering IEEE, pp.178-187, 1978

[19] Coleman, D., D. Ash, B. Lowther, and P. Oman, 'Using metrics to evaluate software system maintainability', IEEE COMPUTER, pp.44-49, 1994

◐ 저 자 소 개 ◑

임 좌 상

1991년 New South Wales University MIS전공 졸업 (석사)
1994년 New South Wales University MIS전공 졸업 (박사)
1997년~현재 상명대학교 디지털미디어학부 교수
관심분야 : MIS, 소프트웨어공학, 소셜네트워크서비스, 감성콘텐츠.
E-mail : jslim@smu.ac.kr

김 진 만

2003년 한서대학교 수학과 졸업 (학사)
2006년 상명대학교 디지털미디어대학원 정보통신학과 졸업 (석사)
2008년~현재 상명대학교 일반대학원 컴퓨터과학과 박사과정
관심분야 : 소프트웨어공학, 소셜네트워크서비스, 감성콘텐츠
E-mail : hansumo81@gmail.com