DOI: 10.3745/KIPSTD.2009.16-D.4.527

클래스 연산의 선행/후행 조건에 바탕을 둔 클래스의 상태 다이어그램 자동 구성 기법

이 광 민[†] · 배 정 호[††] · 채 흥 석[†††]

요 약

상태 다이어그램은 객체가 가질 수 있는 상태와 각 상태에서 수행 할 수 있는 전이를 사용하여 객체의 동적행위를 표현한다. 본 논문에서는 클래스 연산들의 선행/후행 조건들을 이용하여 상태다이어그램을 자동으로 생성하는 기법을 제안한다. 그리고 본 논문에서 제시한 기법을 구현하는 도구 SDAG(State Diagram Automatic Generation Tool)을 제작하였다. 추가적으로 생성된 상태 다이어그램의 복잡도를 감소시키기 위하여 연산의 종류를 고려한 상태 다이어그램 생성 방법과 생성된 다이어그램을 측정하는 방법을 제안하고 실험하였다.

키워드 : 객체지향 프로그래밍, 선행/후행 조건, 상태 다이어그램

An Automatic Construction Approach of State Diagram from Class Operations with Pre/Post Conditions

Lee Kwang-Min[†] · Jung Ho Bae[††] · Heung Seok Chae[†††]

ABSTRACT

State diagrams describe the dynamic behavior of an individual object as a number of states and transitions between these states. In this paper, we propose an automated technique to the generation of a state diagram from class operations with pre/post conditions. And I also develop a supporting tool, SDAG (State Diagram Automatic Generation tool). Additionally, we propose a complexity metric and a state diagram generation approach concerning types of each operation for decreasing complexity of generated state diagram.

Keywords : Object-Oriented Programming, Pre/Post Condition, State Diagram

1. 서 론

클래스는 객체지향 시스템에서 시스템의 구성 기본 단위이다. 즉 개발자는 요구사항을 바탕으로 적절한 클래스 및 그 구성 요소(속성과 연산)와 클래스 간의 관계(연관, 일반화, 의존 등)를 도출하며, 시스템은 이들 클래스로부터 생성된(instantiated) 객체(objects)들 간의 상호 작용(interaction)을 통하여 동작한다.

하나의 클래스에 대해서는 정적 모델과 동적 모델을 모두 정의할 수 있다. 클래스의 정적 모델은 클래스에 속한 속성(attribute)과 연산(operation)으로 구성된다. 클래스 한 객체의 동적 행위는 상태 다이어그램(state diagram)으로 표현할 수 있고, 상태 다이어그램으로 표현된 클래스의 동적 모델은 클래스에 포함된 연산들의 적절한 수행 순서를 명시하고 있다. 상태 다이어그램에서 상태는 현재 객체의 속성들이 어떤 값을 가지는 지를 나타낸다. 그리고 전이(transition)는 전이가 수행 될 수 있는 상태에서 전이를 수행할 수 있는 연산이 입력되면 객체가 어떤 상태로 되는 지를 나타낸다. 객체의 동작은 바로 객체의 클래스에 정의된 연산의 수신에 따른 객체의 반응을 뜻하며, 객체가 생성된 후에 객체가 소멸되기 전까지의 객체가 어떤 순서로 연산들을 수행할 수 있고 각 연산을 어떻게 처리할 수 있는가를 보여준다.

상태 다이어그램을 이용하면 연산들의 수행 순서를 파악할 수 있으므로 요구사항의 확인(validation)[1], 정형적 검증, 테스트 케이스 자동 생성[2-4] 등과 같은 다양한 분야에서 이용되고 있다.

- 테스팅: 상태 다이어그램으로부터 연산들의 수행가능한 모든 순서를 파악할 수 있으므로, 상태 다이어그램을

※ 이 논문은 2007년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 연구되었음(KRF-2007-331-D00410).
† 정 회 원 : (주)사이버맵월드부설연구소
†† 준 회 원 : 부산대학교 컴퓨터공학과 박사과정
††† 정 회 원 : 부산대학교 컴퓨터공학과 조교수(교신저자)
논문접수 : 2009년 1월 16일
수 정 일 : 1차 2009년 3월 3일
심사완료 : 2009년 3월 3일

바탕으로 클래스를 테스팅하는 다양한 연구가 수행되고 있다. 예를 들어, 클래스를 테스팅할 때 상태 다이어그램의 각 상태를 적어도 한 번 씩은 방문하도록 테스트 케이스를 생성한다, 상태 다이어그램의 각 전이를 적어도 한 번 씩은 방문한다, 또는 순환(cycle)을 제외한 시작 상태에서 종료 상태까지의 모든 경로(path)를 테스팅한다와 같은 체계적인 기준이 사용될 수 있다. 이는 상태 의존적인 복수 개의 연산을 가지는 클래스를 테스팅할 때 이론적인 측면에서는 무한한 경로를 테스트해야 하는 문제가 있지만, 상태 다이어그램으로 표현되어 있는 실제 가능한 유한한 개수의 경로만을 조사함으로써 클래스를 테스팅하는 것이 가능하게 된다. 상태에 근거한 테스팅 방법이 효과가 있다는 많은 연구가 보고되고 있다[5-7].

- 정형적 검증: 클래스의 객체 모델은 테스트 케이스의 자동 생성뿐만 아니라, 다양한 정형적 검증의 바탕이 되고 있다. 예를 들어, 상태 기계로 표현된 객체의 행동을 분석하여 실시간성[8], 안전성(safety)[9, 10], 실행시 검증[11] 등과 같은 다양한 분야에 사용되고 있다.
- 코드 생성 및 역공학: 클래스의 정적 모델을 통해서는 클래스의 속성과 연산에 대한 선언만이 코드로서 생성될 수 있다. 반면에 동적 모델을 통해서는 연산의 구현 코드를 전체 또는 부분적으로 자동 생성하는 것이 가능하다[12, 13]. 뿐만 아니라, 복잡한 소스 코드의 이해를 돕기 위해서 소스 코드로부터 동적 모델을 자동으로 추출하는 연구도 진행되고 있다[14-16].

상태 다이어그램과 같은 동적 모델은 이와 같이 많은 분야에서 사용되지만 소프트웨어 개발 과정에서 클래스의 동적 모델을 정의하는 것은 매우 어렵다[17-19]. 이는 동적 모델에서는 개별적인 연산의 기능뿐만 아니라 연산들이 수행 가능한 순서 및 모든 연산 간의 상호 작용에 대한 파악이 필요하기 때문이다. 즉, 복수 개의 연산을 모두 고려하고 모든 연산의 기능과 의미를 이해하여 이들을 모두 반영하는 하나의 상태 다이어그램을 구성하는 작업은 매우 어렵다. 따라서 요구 사항의 확인, 정형적 검증, 테스팅 등의 기술을 적용하기 위해서는 클래스의 상태 다이어그램을 효과적으로 구성하는 방법이 요구된다.

본 논문에서는 다양한 상태 다이어그램의 이용 분야 중에서, 자동 분석 및 자동 처리에 적합한 상태 다이어그램을 자동으로 생성하는 방법을 제시하고, 제안한 방법을 구현하는 자동화 도구를 제작하여 검증한다. 상태 다이어그램 자동 생성은 먼저 클래스 정보를 이용하여 단위 상태 다이어그램 생성 과정을 통해 복수 개의 단위 상태 다이어그램을 생성한다. 그리고 생성된 복수 개의 단위 상태 다이어그램을 단위 상태 다이어그램 합성 과정을 통해 합성된 상태 다이어그램을 생성한다.

그리고 도구에 의해 자동으로 생성되는 상태 다이어그램의 복잡도를 감소시키기 위해 상태 다이어그램 복잡도 메트릭을 제시한다. 제시된 상태 다이어그램 복잡도 메트릭은 단위 상태 다이어그램 생성과 단위 상태 다이어그램 합성 과정에서 더 낮은 복잡도를 가지는 결과를 생성할 수 있도록 한다.

본 논문의 구성은 다음과 같다. 2절에서는 논문의 연구 배경으로서 연산의 선행/후행 조건과 동적 모델을 표현하기 위한 UML의 상태 다이어그램에 대하여 설명한다. 3절에서는 상태 다이어그램을 자동 생성하기 위한 방법에 대해 설명한다. 4절에서는 제시된 방법을 통해 구현된 상태 다이어그램 자동 생성 도구와 적용 사례를 살펴본다. 5절에서는 관련 연구들을 살펴보고 마지막으로 6절에서 결론 및 향후 연구 방향을 기술한다.

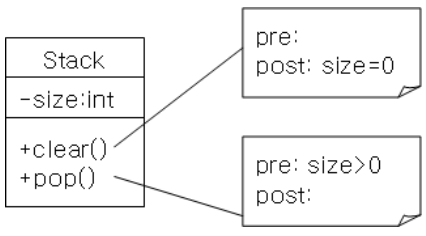
2. 연구 배경

본 절에서는 논문의 연구배경으로서 연산의 선행/후행 조건과 상태 다이어그램을 알아본다. 본 논문에서는 정적 모델 정보로 클래스 연산의 선행/후행 조건 정보를 사용하며, 클래스의 동적 모델을 표현하기 위해 상태 다이어그램을 사용한다. 클래스 연산의 선행/후행 조건에 포함된 정보를 통해 단위 상태 다이어그램을 생성하며, 생성된 단위 상태 다이어그램들을 합성하여 상태 다이어그램을 생성하도록 한다.

2.1 연산의 선행/후행 조건

선행 조건과 후행 조건은 연산의 의미(semantic)를 표현하는 방법으로 객체지향 방법론에서 “계약에 의한 설계(Design by Contract)”라고 불린다[20]. “계약에 의한 설계”는 객체지향 프로그래밍뿐만 아니라 모델링에서도 일반적으로 사용된다. UML(Unified Modeling Language)[21]에서는 OCL(Object Constraint Language)[22]를 이용하여 클래스 각 연산의 선행/후행 조건을 명시적으로 기술하기도 한다. 또한 모델링 단계에서 OCL로 표현된 연산의 선행/후행 조건을 Java 또는 C++ 코드로의 자동 변환도 가능하다[21-28].

선행 조건은 연산이 수행되기 전에 항상 참이어야 하는 조건이다. 만약 선행 조건이 참이 아니라면 연산은 수행 될 수 없다. 반대로, 후행 조건은 연산이 수행되고 나서 항상 참이 되는 조건이다. 예를 들어, (그림 1)의 *Stack*의 *pop()* 연산은 Stack 내부에 하나 이상의 저장 요소가 있을 때에 정상적으로 동작한다. 따라서 *pop()* 연산의 선행 조건은 “*Stack* 내에 하나 이상의 저장 요소가 있을 것”이다. 또한, *Stack*의 *clear()* 연산이 수행된 뒤에는 *Stack* 내부에 어떠한

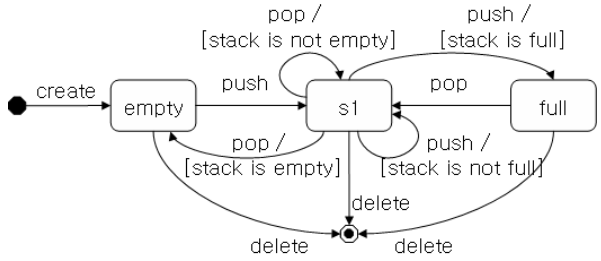


(그림 1) Stack 클래스의 예

저장 요소도 들어있지 않다. 따라서 *clear()* 연산의 후행 조건은 "*Stack* 내에 저장 요소가 없을 것"이다.

2.2 상태 다이어그램

상태 다이어그램은 한 객체의 동적인 특성을 모델링하기 위해 사용되는 UML의 다이어그램들 중 하나이다. 상태 다이어그램은 객체의 생명 주기 동안의 동적 모델 정보를 유한 상태 기계의 형태로 표현한다. (그림 2)는 *Stack* 클래스에 대한 상태 다이어그램의 예를 보여 준다.



(그림 2) Stack 클래스에 대한 상태 다이어그램 예

Stack 클래스에서는 최초에 *create()* 연산이 수행될 수 있으며, 이후에는 *push()* 연산만이 수행될 수 있다. 상태 *s1*에서는 *push()* 연산과 *pop()* 연산이 모두 수행 가능하며, *push()* 연산에 의해서 *full* 상태가 된 후에는 *pop()* 연산만이 허용된다. 여기서 상태 *s1*에서 *push()* 연산은 상태 *s1*로 돌아오는 전이와 상태 *full*로의 전이에 표시되어 있다. 이 경우, *push()* 연산의 후행 조건에 의해 *push()* 연산이 끝난 뒤 내부 스택이 가득 차 있다면 *full* 상태로 전이가 일어나며, 내부 스택이 가득 차 있지 않다면 *s1* 상태로 돌아오도록 전이가 일어난다.

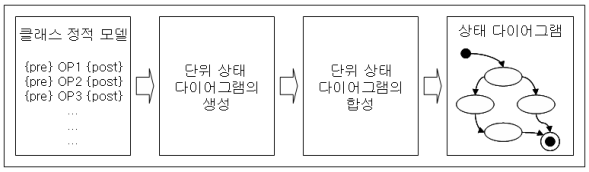
3. 상태 다이어그램 자동 생성

본 절에서는 객체에 포함된 연산의 선행/후행 조건을 이용하여 해당 객체의 상태 다이어그램을 자동적으로 생성하기 위한 방법과 생성할 상태 다이어그램을 개선하기 위한 방법을 알아본다. 3.1 절에서는 상태 다이어그램을 생성하는 전체 생성 흐름을 간략하게 알아보고, 3.2 절과 3.3 절에서는 전체 생성 흐름 중 가장 중요한 두 가지 과정에 대해 알아본다. 3.4 절에서는 3.2 절과 3.3 절의 생성 흐름을 예제를 통해 확인한다. 그리고 3.5절에서는 상태 다이어그램의 자동 분석을 위해 상태 다이어그램의 복잡도를 개선하기 위한 두 가지 복잡도 개선 방안에 대해 알아본다.

3.1 상태 다이어그램 생성 방법

단위 상태 다이어그램은 클래스의 한 연산으로부터 구성된 상태 다이어그램을 의미한다. 클래스의 연산은 연속해서 여러 번 호출될 수 있으므로 하나의 연산만으로도 상태 다이어그램을 구성할 수 있다. 본 연구에서는 먼저 클래스의 각 연산 별로 단위 상태 다이어그램을 구성한다. 그리고 구성된 개별적인 단위 상태 다이어그램을 합성함으로써 클래스 전체 연산의 상태 다이어그램을 유도하는 방법을 사용하려고 한다.

(그림 3)은 클래스 정적 모델, 즉 선행/후행 조건이 추가된 연산 명세 정보로부터 동적 모델(상태 다이어그램)을 자동 생성하는 과정을 보여 준다. 동적 모델 자동 생성은 1) 단위 상태 다이어그램의 생성, 2) 단위 상태 다이어그램의 합성으로 구성된다.



(그림 3) 상태 다이어그램 생성 흐름

1) 단위 상태 다이어그램의 생성

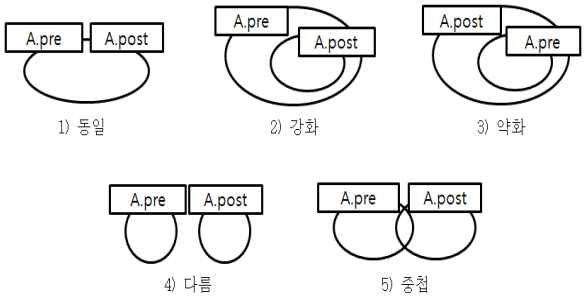
단위 상태 다이어그램의 생성 단계에서는 클래스의 각 연산마다 단위 상태 다이어그램을 생성한다. 단위 상태 다이어그램은 연산의 선행/후행 조건에 따라 다른 형태로 생성된다. 각 단위 상태 다이어그램은 단 하나의 연산을 수행하는 객체의 상태 다이어그램을 의미한다.

2) 단위 상태 다이어그램의 합성

단위 상태 다이어그램의 합성 단계에서는 클래스의 연산마다 생성된 전체 단위 상태 다이어그램을 순차적으로 합성한다. 합성된 상태 다이어그램은 단위 상태 다이어그램의 전이에 존재하는 선행/후행 조건에 따라 다른 형태로 합성된다.

3.2 단위 상태 다이어그램의 생성

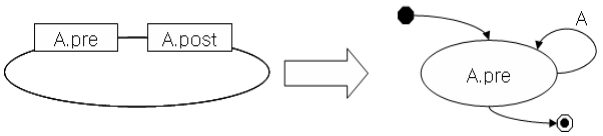
단위 상태 다이어그램의 생성은 먼저 선행/후행 조건 간의 관계에 따라 5가지 유형으로 나눌 수 있다. 서로 다른 두 조건은 완전히 일치하거나(동일), 어느 하나가 다른 하나를 포함하거나(강화, 약화), 동시에 만족하는 조건이 하나도 없거나(다름), 일부만 동시에 만족하는 조건이 있는 경우 중 하나이다. 연산 A에 대해 선행/후행 조건을 각각 A.pre와 A.post라고 한다면 (그림 4)는 선행/후행 조건 간의 5가지 유형을 보여 준다.



(그림 4) 연산의 선행/후행 조건의 5가지 포함 관계 유형

1) 동일: 선행 조건과 후행 조건이 일치하는 경우

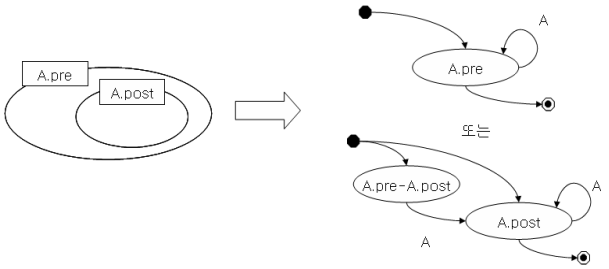
두 조건이 동일 관계인 경우 연산 A의 상태 다이어그램은 (그림 5)와 같이 생성된다. 연산 A는 항상 반복 가능하다. 단위 상태 다이어그램은 선행 조건 혹은 후행 조건을 기준으로 한 상태를 생성한 뒤 상태에 대한 반복적 연산으로 정의된다.



(그림 5) 동일 관계

2) 강화: 선행 조건이 후행 조건을 포함하는 경우

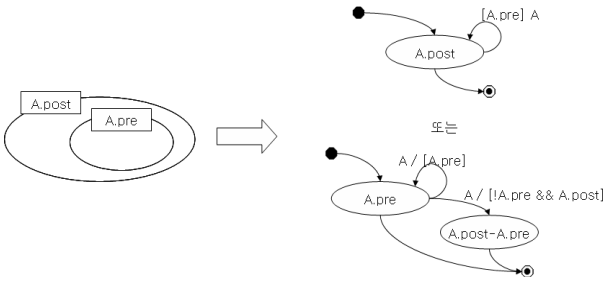
두 조건이 강화 관계인 경우 연산 A의 단위 상태 다이어그램은 (그림 6)에 제시된 두 상태 다이어그램 중 하나로 생성된다. 연산 A는 항상 반복 가능하다. 따라서 선행 조건을 기준으로 한 상태를 생성한 뒤 상태에 대한 반복적 연산으로 정의될 수 있고, 혹은 후행 조건에 포함되지 않는 선행 조건 상태를 생성한 뒤 후행 조건 상태로 향하는 연산과, 후행 조건 상태에 대한 반복적 연산으로 정의된다.



(그림 6) 강화 관계

3) 약화: 후행 조건이 선행 조건을 포함하는 경우

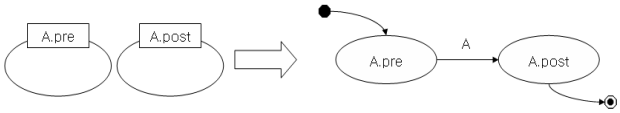
두 조건이 약화 관계인 경우 연산 A의 단위 상태 다이어그램은 (그림 7)에 제시된 두 상태 다이어그램 중 하나로 생성된다. 연산 A는 선행 조건을 만족하면 반복할 수 있다. 따라서 후행 조건을 기준으로 한 상태를 생성한 뒤 선행 조건에 따라 수행 가능한 반복적 연산으로 정의될 수 있고, 혹은 선행 조건을 기준으로 한 상태와 선행 조건을 포함하지 않는 후행 조건을 기준으로 한 상태를 생성한 뒤 후행 조건에 따라 수행 가능한 반복적 연산으로 정의될 수 있다.



(그림 7) 약화 관계

4) 다름: 선행 조건과 후행 조건이 아무런 연관이 없는 경우

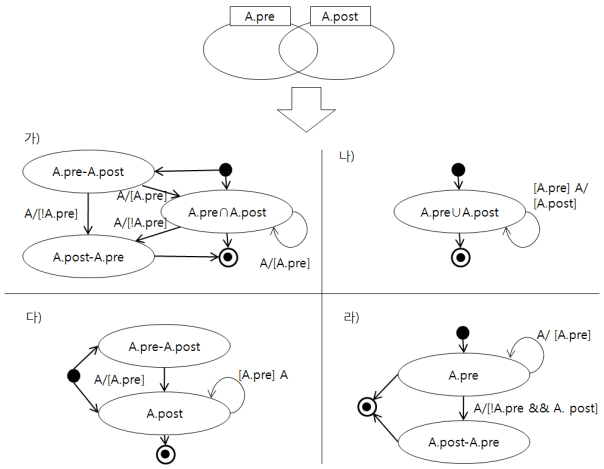
두 조건이 다른 관계인 경우 연산 A의 단위 상태 다이어그램은 (그림 8)과 같이 생성된다. 연산 A는 선행 조건을 기준으로 한 상태에서 후행 조건을 기준으로 한 상태로 향하는 연산으로 정의될 수 있다.



(그림 8) 다른 관계

5) 중첩: 선행 조건과 후행 조건이 일부 연관이 있는 경우

두 조건이 중첩 관계인 경우 연산 A의 단위 상태 다이어그램은 (그림 9)에 제시된 네 가지 상태 다이어그램 중 하나로 생성된다.



(그림 9) 중첩 관계

가) 후행 조건을 제외한 선행 조건을 기준으로 한 상태와, 선행 조건을 제외한 후행 조건을 기준으로 한 상태, 그리고 선행/후행 조건을 모두 만족하는 상태로 생성한다. 즉, 각 조건집합의 부분별로 상태를 생성한다. 그리고 각 상태에서 알맞은 후행 조건을 만족하는 경우에 수행 가능한 반복적 연산으로 정의

나) 동일 관계와 같이 생성하고 전이의 선행/후행 조건을 수정

다) 강화 관계와 같이 생성하고 전이의 선행/후행 조건을 수정

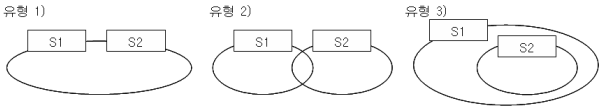
라) 약화 관계와 같이 생성하고 전이의 선행/후행 조건을 수정

두 조건이 강화 또는 약화 또는 중첩 관계일 때는 두 가지 이상의 상태 다이어그램이 생성 될 수 있다. 각 관계의 서로 다른 모양의 상태 다이어그램들은 같은 행위를 수행하지만 상태의 수나 전이의 수, 전이의 선행/후행 조건이 다르다. 즉, 각 상태 다이어그램들 중 어떠한 모양의 상태 다이어그램을 생성하는지에 따라서 각 연산의 단위 상태 다이어

그램들의 복잡도가 달라진다. 이는 단위 상태 다이어그램 합성 단계를 거쳐 최종 생성된 상태 다이어그램의 복잡도에도 영향을 미친다. 각 연산의 단위 상태 다이어그램을 생성할 때에 어떤 모양의 상태 다이어그램을 생성할지는 생성 후의 단위 상태 다이어그램의 복잡도를 고려하여 결정할 수 있다.

3.3 단위 상태 다이어그램의 합성

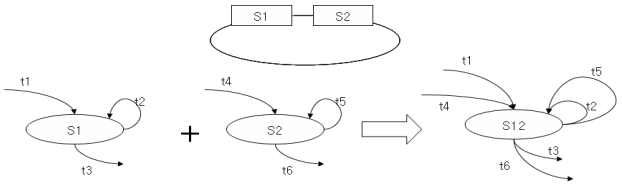
단위 상태 다이어그램의 합성은 앞 단계에서 생성된 상태들을 합성하고, 상태에 연결된 전이를 합성의 결과에 따라 조합/분화함으로써 진행된다. 단위 상태 다이어그램의 합성은 상태의 상태 집합의 관계에 따라 세 가지 유형으로 나눌 수 있다. 두 상태를 각각 S1과 S2라고 할 때, 두 상태 집합의 포함 관계는 (그림 10)과 같다.



(그림 10) 두 상태 집합의 3가지 포함 관계 유형

• 유형 1) 두 상태의 상태 집합이 동일한 경우

두 상태 S1과 S2의 상태 집합이 동일하다면, 상태 합성 과정은 (그림 11)과 같다. 상태 S1 및 S2와 동일한 상태 집합을 가지는 상태 S12가 결과 상태로서 생성되고, 각 상태에 연결된 전이들은 모두 결과 상태에 포함된다.



(그림 11) 두 상태의 상태 집합이 동일한 경우 합성 방법

• 유형 2) 두 상태의 상태 집합 사이에 일부 연관이 있는 경우

두 상태 S1과 S2의 상태 집합 사이에 일부 연관이 있다면, 상태 합성 과정은 (그림 12)와 같다. (그림 13)에서, 두 상태 S1과 S2의 상태 집합의 교집합의 상태 집합을 가지는 상태를 S1 ∩ S2, 합집합의 상태 집합을 가지는 상태를 S1 ∪ S2, 차집합의 상태 집합을 가지는 상태를 각각 S1-S2와 S2-S1이라고 정의한다. 상태 다이어그램의 합성은 합성 결과 상태들의 다음의 네 가지 경우 중 하나로 합성된다.

가) 상태 S1과 S2의 상태 집합의 교집합을 상태 집합으로 하는 상태 S1 ∩ S2를 만들고, 상태 S1과 S2에 연결된 전이들 가운데 선행/후행 조건이 만족되는 전이를 연결

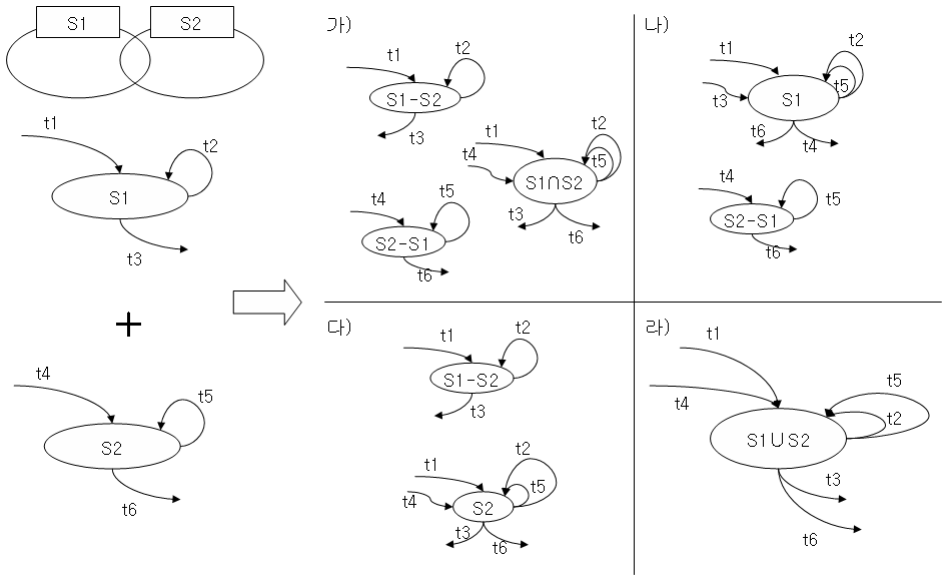
나) 상태 S1과 동일한 상태 집합과, 상태 S2에서 상태 S1을 제외한 상태 집합을 가지는 상태 S2-S1을 만들고, 상태 S2에 연결된 전이들 가운데 선행/후행 조건이 만족되는 전이를 상태 S1에 연결

다) 상태 S2와 동일한 상태 집합과, 상태 S1에서 상태 S2를 제외한 상태 집합을 가지는 상태 S1-S2를 만들고, 상태 S1에 연결된 전이들 가운데 선행/후행 조건이 만족되는 전이를 상태 S2에 연결

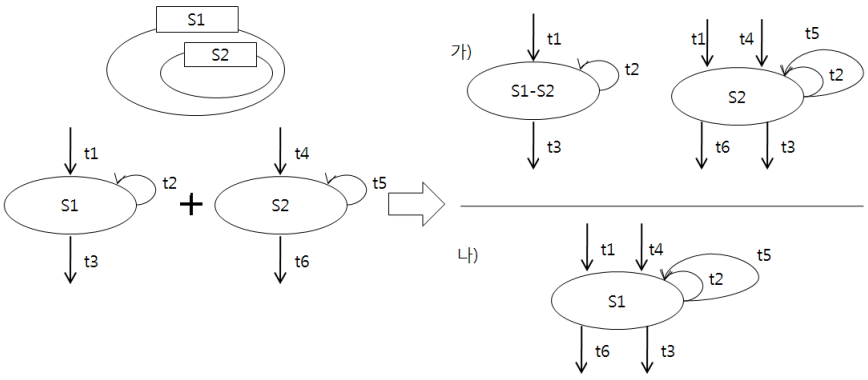
라) 상태 S1과 상태 S2의 상태 집합의 합집합을 상태 집합으로 하는 상태 S1 ∪ S2를 만들고 양쪽 상태에 연결된 전이를 연결

• 유형 3) 하나의 상태의 상태 집합이 다른 상태의 상태 집합을 포함하는 경우

하나의 상태의 상태 집합이 다른 상태의 상태 집합을 포함한다면, 상태 합성 과정은 (그림 13)과 같다. (그림 13)에서, 상태 S1의 상태 집합에서 상태 S2의 상태 집합을 뺀 집합을 상태 집합으로 하는 상태를 S1-S2라고 정의한다. 상태



(그림 12) 두 상태의 상태 집합 사이에 일부 연관이 있는 경우 합성 방법



(그림 13) 어느 상태 집합이 다른 상태 집합을 포함하는 경우 합성 방법

집합은 합성 결과 상태의 복잡도에 따라 다음의 두 가지 경우 중 하나로 합성된다.

가) 상태 S1의 상태 집합에서 상태 S2의 상태 집합을 뺀 집합을 상태 집합으로 하는 상태 S1-S2를 생성하여 상태 S1에 연결된 전이들 중 선행/후행 조건이 만족되는 전이를 연결하고, 상태 S2에 상태 S1에 연결된 전이들 중 선행/후행 조건이 만족되는 전이를 연결

나) 상태 S2에 연결된 전이를 모두 상태 S1에 연결

3.4 단위 상태 다이어그램 생성/합성 예제

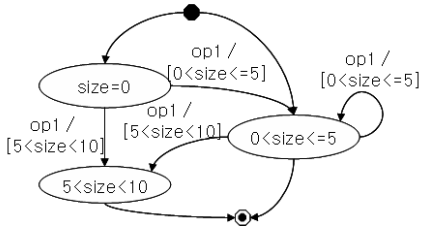
본 절에서는 (그림 14)의 클래스 A의 선행/후행 조건을 가지는 연산 op1과 op2를 이용하여 단위 상태 다이어그램을 생성하고 생성된 단위 상태 다이어그램을 합성하는 예제를 보여준다. 첫 번째 연산 op1은 선행 조건이 0<=size<=5 이고, 후행 조건이 0<size<10이다. 두 번째 연산 op2는 선행 조건이 0<=size<3 이고, 후행 조건이 size=10 이다.

클래스 A	
연산이름	선행조건
	후행조건
op1	0 <= size <= 5
	0 < size < 10
op2	0 <= size < 3
	size = 10

(그림 14) 클래스 A

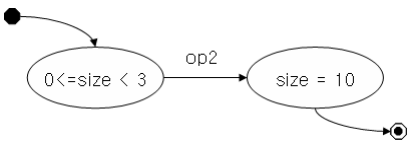
3.4.1 단위 상태 다이어그램 생성 예제

클래스의 A의 첫 번째 연산 op1에 대한 단위 상태 다이어그램은 (그림 15)와 같다. op1의 선행 조건과 후행 조건은 중첩 관계이며 이 예에서는 중첩 관계일 때 생성 될 수 있는 4 종류의 단위 상태 다이어그램 중 가)를 이용한다. op1의 단위 상태 다이어그램은 size=0, 0<size<=5, 5<size<10의 상태 집합을 가지는 3개의 상태와 7개의 전이로 구성된다.



(그림 15) op1의 단위 상태 다이어그램

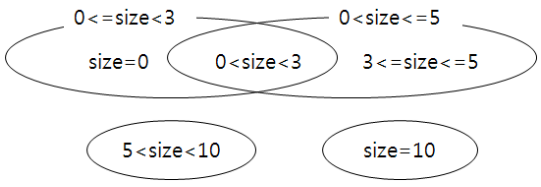
두 번째 연산 op2에 대한 단위 상태 다이어그램은 (그림 16)과 같다. op2의 선행 조건과 후행 조건은 다른 관계이므로 각 조건을 상태 집합으로 가지는 2개의 상태와 1개의 전이로 구성되는 단위 상태 다이어그램이 생성된다.



(그림 16) op2의 단위 상태 다이어그램

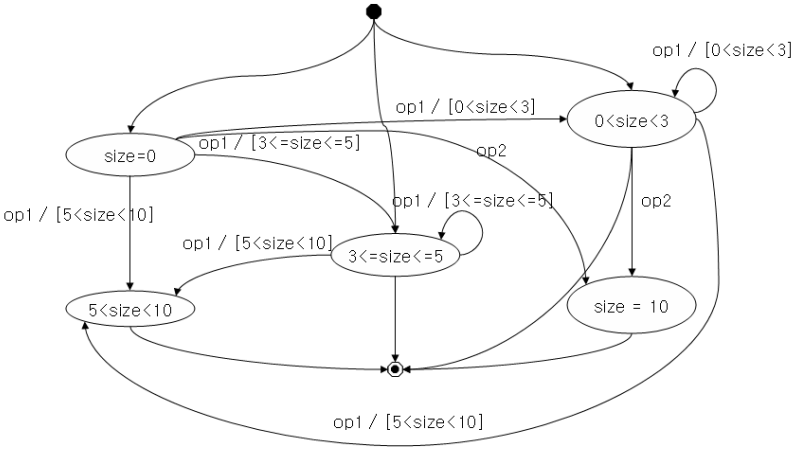
3.4.2 단위 상태 다이어그램 합성 예제

단위 상태 다이어그램들의 상태들 중 3.3절에 설명된 유형 1), 유형 2), 유형 3)에 해당하는 상태들에 대해서 합성하고, 전이를 연결하여 단위 상태 다이어그램들을 합성한다. op1의 단위 상태 다이어그램의 size=0 상태와 op2의 단위 상태 다이어그램의 0<=size<3의 상태의 관계는 유형 3)에 해당한다. 그리고 0<size<=5와 0<=size<3은 유형 2)에 해당한다. 이 상태들의 관계를 벤 다이어그램으로 나태면 (그림 17)과 같다.



(그림 17) 각 상태 집합들 간의 관계

각 상태들의 상태 집합간의 관계를 고려하여 op1과 op2의 단위 상태 다이어그램을 합성하면 (그림 18)과 같다. op1과 op2의 최종 상태 다이어그램은 모두 5개의 상태를 가지



(그림 18) 클래스 A의 최종 상태 다이어그램

며 15개의 전이를 가진다. 최종 상태 다이어그램의 상태의 수와 각 단위 상태 다이어그램의 상태의 수의 합은 같지만, op1의 상태 size=0, 0<size<=5와 op2의 상태 0<=size<3가 합성되어 최종적으로 size=0, 0<size<3, 3<=size<=5, 5<size<10, size=10의 상태들을 가짐을 알 수 있다.

3.5 복잡도 개선 방안

최종적으로 생성된 다이어그램은 연산의 수가 커지면 커질수록 상태의 수가 증가 할 가능성도 커지며, 단위 상태 다이어그램을 어떤 방법으로 생성하는지에 따라서 복잡도가 달라질 수 있다. 이 절에서는 자동 분석의 입장에서 복잡도를 감소시키기 위해 두 가지 방안을 제시한다.

3.5.1 연산의 유형 고려

클래스의 연산은 각 연산의 속성에 따라 생성자(Constructor), 소멸자(Destructor), 액세서(Accessor) 연산, 뮤테이터(Mutator) 연산의 네 가지 유형으로 구분할 수 있다.

- 생성자: 객체를 생성할 때 사용되는 연산을 의미한다. 생성된 뒤의 객체는 다른 연산이 수행되기 전에 특정한 상태로 존재할 수 있다. 생성 연산의 후행 조건의 상태는 상태 다이어그램에서 시작 상태로부터의 전이를 가지는 상태로 표현될 수 있다.
- 소멸자: 객체가 소멸될 때 사용되는 연산을 의미한다. 소멸 연산의 정보를 통해 객체의 소멸이 수행되기 전의 특정한 상태를 확인할 수 있다. 소멸 연산의 선행 조건의 상태는 상태 다이어그램에서 종료 상태로의 전이를 가지는 상태로 표현될 수 있다.
- 액세서 연산: 객체의 정보를 수정하거나 변경하지 않고, 객체의 정보를 얻어오기만 하는 연산을 의미한다. 액세서 연산이 호출되었을 때에는 객체의 정보가 변경되지 않기 때문에, 액세서 연산은 상태 다이어그램에서 언제나 자기 자신으로의 전이의 형태로 표현될 수 있다.
- 뮤테이터 연산: 객체의 정보를 바꾸는 연산을 의미한다. 뮤테이터 연산은 선행/후행 조건에 따라 객체의 상태를 바꿀 수 있으므로, 단위 상태 다이어그램 생성 및 합성 단계를 거친다.

연산의 유형을 고려한 전체 상태 다이어그램 생성 흐름은 다음과 같다.

1) 뮤테이터 연산에 의한 단위 상태 다이어그램 생성
2) 생성된 단위 상태 다이어그램 합성
3) 생성자, 소멸자, 액세서 연산에 의한 전이를 추가

이러한 과정은 연산의 유형을 고려하지 않은 모든 연산에 대한 단위 상태 다이어그램 생성 및 합성 과정보다 간단하며, 연산의 유형에 의해 생성, 소멸, 액세서 연산에 의한 전이를 특별한 형태로 제한할 수 있으므로, 생성되는 상태 다이어그램의 복잡도를 줄일 수 있다.

3.5.2 상태 다이어그램의 복잡도 메트릭 활용

상태 다이어그램의 복잡도를 측정하면, 단위 상태 다이어그램 생성과 단위 상태 다이어그램 합성 단계에서 선택할 수 있는 여러 가지의 형태의 상태 다이어그램 중에서 가능한 간단한 상태 다이어그램을 생성하거나 합성할 수 있다. 상태 다이어그램의 복잡도는 단위 상태 다이어그램을 생성할 때와 단위 상태 다이어그램을 합성할 때 사용된다.

- 단위 상태 다이어그램 생성: 단위 상태 다이어그램을 생성할 때, 상태를 나누는 경우에 따라 다양한 상태 다이어그램을 생성할 수 있다. 생성될 단위 상태 다이어그램의 복잡도를 계산하여 가장 낮은 복잡도를 가지는 단위 상태 다이어그램을 생성한다.
- 단위 상태 다이어그램의 합성: 단위 상태 다이어그램을 합성할 때, 상태를 그대로 유지하거나, 두 개의 상태 집합을 합한 상태를 생성하거나, 두 개의 상태 집합의 교집합을 분리할 수 있다. 생성될 결과 다이어그램의 복잡도를 계산하여 가장 낮은 복잡도를 가지는 상태 다이어그램이 되도록 합성한다.

본 논문에서는 상태 다이어그램의 복잡도를 측정하기 위하여 상태의 수(NS), 전이의 수(NT), CC(Cyclomatic Complexity)[29]와 본 논문에서 새롭게 제안하는 SDC(State Diagram Complexity)를 사용한다. 각 복잡도 측정 메트릭의 식은 <표 1>과 같다.

〈표 1〉 복잡도 측정 메트릭

복잡도 측정 메트릭	메트릭 정의
상태의 수(NS)	총 상태의 수
전이의 수(NT)	총 전이의 수
CC(Cyclomatic Complexity)	NT-NS+2
SDC (State Diagram Complexity)	$\frac{\sum C_T(T_i)}{NT}$

- 상태의 수(NS): 상태 다이어그램에 포함된 상태의 수로 정의된다. 시작 상태 및 종료 상태도 포함한다.
- 전이의 수(NT): 상태 다이어그램에 포함된 전이의 수로 정의된다. 시작 전이 및 종료 전이를 포함하며, 자기 자신으로 돌아오는 반복 전이도 포함한다.
- CC (Cyclomatic Complexity)
  (전체 전이의 수 - 전체 상태의 수 + 2)로 정의된다. CC 메트릭은 단일 모듈 복잡도 혹은 시스템 복잡도를 계산하기 위해 정의되었으나, [30]에서 UML 상태 다이어그램의 구조적 복잡도를 측정할 수 있도록 개량되었다.
- SDC (State Diagram Complexity)
  전이의 선행/후행 조건이 복잡하다면 자동 분석에 더 많은 비용이 필요하기 때문에, 자동 분석에서 상태 다이어그램의 복잡도는 전이의 선행/후행 조건과 연관이 있다. 따라서 모든 전이의 선행/후행 조건의 복잡도를 측정하여 상태 다이어그램의 복잡도를 정의해야 할 필요가 있다. 본 논문에서는 자동 분석 측면에서 상태 다이어그램의 복잡도를 측정하기 위해 SDC를 제안한다. 전체 상태 다이어그램의 복잡도 SDC값은 상태 다이어그램 내의 모든 전이 T의 복잡도 $C_T(T)$의 평균으로 정의한다.

$$\frac{\sum C_T(T_i)}{NT}$$

$C_T(T)$: 전이 T의 복잡도로 정의한다. 전이 T의 복잡도는 선행 조건의 복잡도와 후행 조건의 복잡도의 합으로 정의한다.

$C_T(T_i) = C(T_i.< ondition) + C(T_i.postcondition)$

$C(condition)$: 선행 조건 혹은 후행 조건의 복잡도로 정의한다. 자동 분석의 기준에서 선행/후행 조건의 복잡도는 조건의 분석 시간으로 생각할 수 있다. 조건은 여러 BOOLEAN항의 AND, OR, NOT에 의한 조합으로 이루어져 있으며, 각 BOOLEAN항의 분석 시간이 동일하다고 가정한다면 unary 연산자의 복잡도를 1로, binary 연산자의 복잡도를 2로 둘 수 있다. 따라서 본 논문에서는 NOT의 가중치를 1로, AND 및 OR의 가중치를 2로 정하여 조건 내에서 사용되는 횟수에 따라 복잡도를 측정한다.

4. 상태 다이어그램 자동 생성 도구

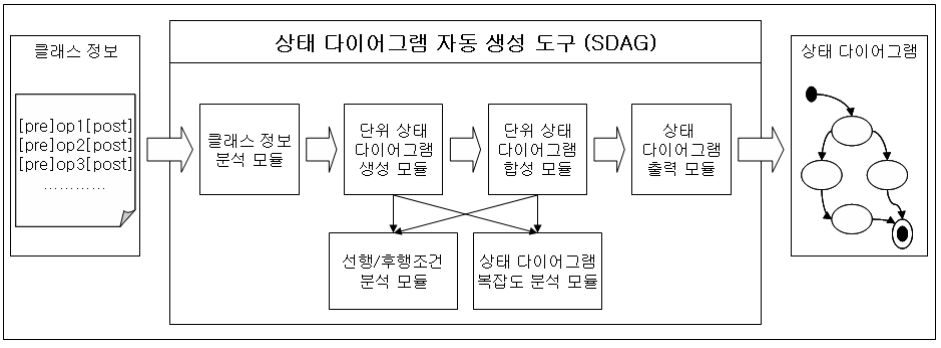
본 장에서는 상태 다이어그램을 자동으로 생성하기 위한 상태 다이어그램 자동 생성 도구 SDAG(State Diagram Automatic Generation tool)를 소개하고, SDAG에 의해 생성된 상태 다이어그램의 복잡도를 분석한다. 4.1절에서는 SDAG의 논리적, 물리적 아키텍처를 살펴보고, 4.2절에서는 사례 연구를 통해 생성된 상태 다이어그램의 복잡도를 분석한다.

4.1 상태 다이어그램 자동 생성 도구의 아키텍처

SDAG는 연산의 선행/후행 조건이 기술된 클래스 정보를 입력받아, 상태 다이어그램을 출력한다. 본 논문에서 제시하는 SDAG의 논리적 아키텍처는 (그림 19)와 같다.

SDAG은 선행/후행 조건이 기술된 클래스의 정보를 입력받아 클래스 정보 분석 모듈에서 클래스의 정보를 분석한다. 그리고 단위 상태 다이어그램 생성 모듈에서 각 연산 별로 단위 상태 다이어그램을 생성하고, 생성된 단위 상태 다이어그램을 단위 상태 다이어그램 합성 모듈에서 합성하여 최종적으로 생성된 상태 다이어그램을 상태 다이어그램 출력 모듈을 이용하여 출력하는 기능들을 수행한다. 단위 상태를 생성하고 합성하는 과정에서 연산의 선행/후행 조건들의 포함 관계를 분석하기 위하여 선행/후행조건 분석 모듈을 사용하고, 생성된 상태 다이어그램의 복잡도를 측정하고 분석하기 위하여 상태 다이어그램 복잡도 분석 모듈을 사용한다.

- 클래스 정보 분석 모듈은 클래스 정보를 읽어 클래스의 연산 유형을 분석한다. 분석된 클래스 정보를 단위 상태 다이어그램 생성 모듈로 보낸다.



(그림 19) SDAG의 논리적 아키텍처

- 단위 상태 다이어그램 생성 모듈은 분석된 클래스 정보의 연산 유형과 상태 다이어그램 복잡도를 고려하여 뮤테이터 연산의 단위 상태 다이어그램을 생성하고 단위 상태 다이어그램 합성 모듈로 보낸다.
- 단위 상태 다이어그램 합성 모듈은 상태 다이어그램 복잡도를 고려하여 단위 상태 다이어그램을 합성한다. 모든 단위 상태 다이어그램을 합성한 뒤, 생성자, 소멸자, 액세서 연산의 전이를 합성하고 상태 다이어그램 출력 모듈로 보낸다.
- 상태 다이어그램 출력 모듈은 단위 상태 다이어그램 합성의 결과로 만들어진 상태 다이어그램 정보를 이용하여 상태 다이어그램을 출력한다.
- 선행/후행조건 분석 모듈은 단위 상태 다이어그램 생성 단계 및 단위 상태 다이어그램 합성 단계에서 연산 및 전이의 선행/후행조건 관계를 분석한다.
- 상태 다이어그램 복잡도 분석 모듈은 단위 상태 다이어그램 생성 단계 및 단위 상태 다이어그램 합성 단계에서 상태 다이어그램의 복잡도를 계산한다.

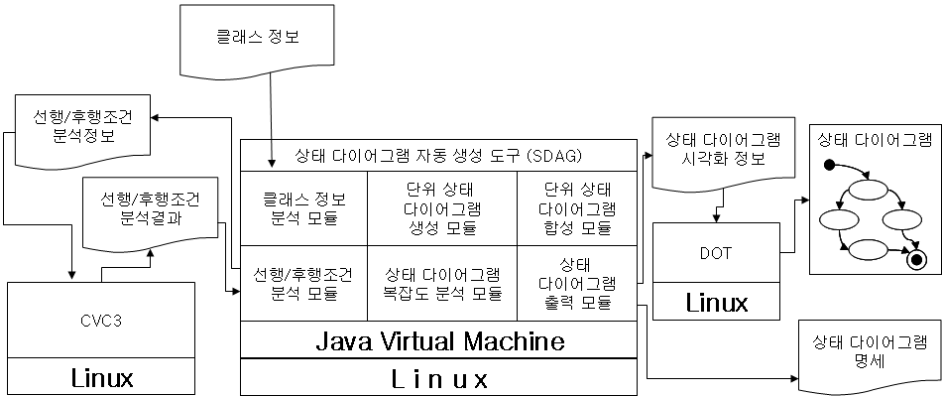
SDAG이 수행되는 물리적 아키텍처는 (그림 20)과 같다. SDAG는 Java를 이용하여 구현하였으며 선행/후행조건 분석 모듈에서 사용하는 이론 증명기(theorem prover)는 CVC3[31]를 사용하였다. 그리고 생성된 상태 다이어그램을 출력하기 위하여 DOT[32]을 이용하였다.

CVC3는 정량자(quantifies), 부분 함수(partial functions), 그리고 술어 하위 타입들을 포함한 내제된 이론들을 이용하여 다양한 종류의 일계논리(first-order logic)의 정당성을 자동으로 검사하는 도구이다. 각 선행/후행 조건의 포함 관계에 대한 참/거짓 질의를 CVC3에 입력하여, 조건들 간의 포함 관계를 평가할 수 있다. OCL을 통해 선행/후행 조건을 표현할 수는 있으나, OCL로 표현된 일계논리를 평가하는 자동화된 도구는 현재까지 나와있지 않다. 따라서 여기서는 선행/후행 조건을 수식의 형태로 입력하여 CVC3를 통하여 분석한다. DOT는 그래프 정보 텍스트를 읽어들여 방향 그래프를 알맞은 형태로 그리는 공개 도구이다. SDAG의 상태 다이어그램 출력 모듈은 상태 다이어그램 시각화 정보를 출력한 뒤 DOT를 사용하여 상태 다이어그램을 시각화한다.

4.2 적용 사례

본 절에서는 총 8개의 클래스를 JML(Java Modeling Language)[33]의 클래스 견본에서 추출하여 상태 다이어그램을 자동 생성하였다. JML은 자바 모듈의 행위를 명세하기 위해 사용되는 행위 인터페이스 명세 언어이다. JML에는 자바 클래스 연산의 선행/후행 조건을 기술할 수 있게 되어 있다. 사례 연구를 위해 사용된 클래스들의 연산 유형별 연산의 수는 <표 2>와 같다.

<표 3>은 SDAG를 사용하여 자동 생성된 8개 클래스의 상태 다이어그램 생성 방법에 따른 상태 다이어그램의 복잡도 정보이다. 상태 다이어그램을 기본 생성 방식과, 연산의 유형만을 고려한 방식, 다이어그램의 복잡도만 고려한 방식, 연산의 유형과 복잡도를 함께 고려한 방식으로 생성하였을 때의 복잡도 값을 확인할 수 있다.



(그림 20) SDAG의 물리적 아키텍처

〈표 2〉 각 클래스의 연산 유형별 연산의 수

클래스 이름	전체 연산 수	생성자 수	소멸자 수	액세서 연산 수	뮤테이터 연산 수
a. Account	5	1	0	3	1
b. ATM	10	1	0	3	6
c. BankCard	5	1	0	3	1
d. BlankReader	4	1	1	0	2
e. BufferdReader	3	0	1	1	1
f. MoneyOps	11	0	0	8	3
g. PriorityQueue	6	1	0	2	3
h. Reader	2	0	1	0	1

〈표 3〉 상태 다이어그램 생성 방법에 따른 각 클래스의 상태 다이어그램 복잡도

클래스	기본 생성				연산 유형 고려				복잡도 고려				연산 유형 및 복잡도 고려			
	NS	NT	CC	SDC	NS	NT	CC	SDC	NS	NT	CC	SDC	NS	NT	CC	SDC
a	4	14	12	83	4	12	10	64	3	9	8	75	4	12	10	64
b	9	170	163	580	9	56	49	375	3	19	18	478	5	24	21	303
c	4	18	16	66	4	11	9	49	3	7	6	55	4	11	9	49
d	8	40	34	1667	8	38	32	1205	4	11	9	1122	4	10	8	855
e	5	16	13	240	5	12	9	232	4	10	8	215	5	12	9	232
f	4	42	40	5	4	26	24	5	3	16	15	9	4	26	24	5
g	5	26	23	35	4	12	10	14	3	14	13	39	4	12	10	14
h	5	10	7	86	5	10	7	86	5	10	7	86	5	10	7	86

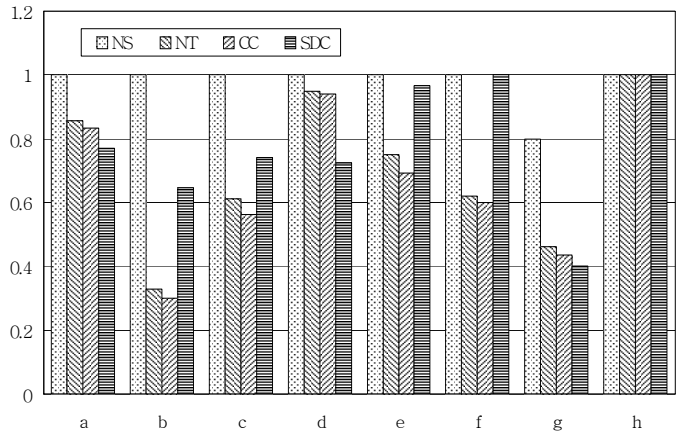
● 연산 유형 고려 방식

(그림 21)은 상태 다이어그램 기본 생성 방법을 1로 하여 연산 유형을 고려하여 상태 다이어그램을 생성하였을 때 각 복잡도 측정 기준에 따른 복잡도를 비교한 그래프이다. 연산의 유형을 고려하면 기본 생성 방법에 비해 전이의 수가 감소하며 복잡도가 감소한다.

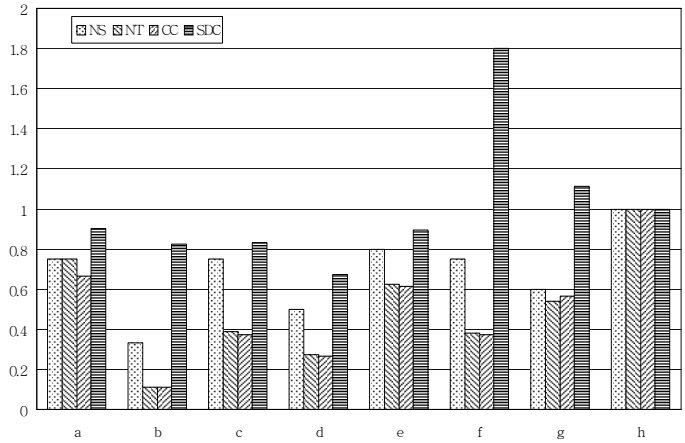
클래스 f의 경우, 전이의 복잡도가 감소하지 않아 상태 다이어그램 복잡도가 감소하지 않았다. 클래스 h의 경우 전이의 수가 감소하지 않았다. 이는 클래스 h에는 하나의 소멸자가 존재하며, 소멸자가 전체 상태에 적용되므로 연산의 유형이 고려되지 않은 것과 동일하기 때문에 상태 다이어그램의 복잡도가 감소하지 않았다. 그 외 6 개의 클래스에 대해 상태 다이어그램의 전이 수가 감소하였다.

● 복잡도 고려 방식

(그림 22)는 상태 다이어그램 기본 생성 방법을 1로 하여 상태 다이어그램의 복잡도를 고려하여 상태 다이어그램을 생성하였을 때 각 복잡도 측정 기준에 따른 복잡도를 비교한 그래프이다. 상태 다이어그램의 복잡도를 고려하면 상태



(그림 21) 연산 유형을 고려하여 생성한 상태 다이어그램의 복잡도



(그림 22) 복잡도를 고려하여 생성한 상태 다이어그램의 복잡도

의 수와 전이의 수가 함께 감소하며 복잡도가 감소한다.

상태 다이어그램의 복잡도를 고려하면 기본 생성 방법에 비해 상태의 병합이 일어나 전체 상태의 수가 감소한다. 클래스 h의 경우, 기존의 상태 다이어그램의 복잡도가 가장 낮다고 판별하여 상태의 수가 감소하지 않았다. 클래스 h, g의 경우, 상태 다이어그램 합성 단계에서 복잡도가 가장 낮은 상태 다이어그램을 합성하지만, 액세서 연산을 추가하면서 액세서 연산의 선행/후행 조건의 복잡도가 추가되어 SDC 복잡도가 증가하였다. 그 외 5 개의 클래스에 대해 상태 다이어그램의 복잡도 값이 감소한다.

- 연산 유형 및 복잡도 고려 방식

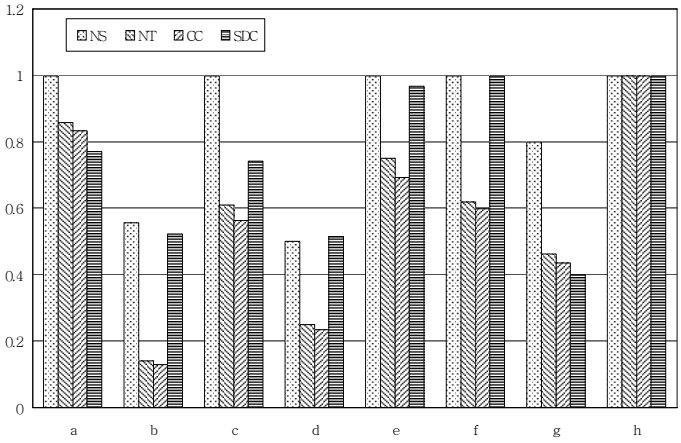
(그림 23)은 상태 다이어그램 기본 생성 방법을 1로 하여 연산 유형과 상태 다이어그램의 복잡도를 함께 고려하여 상태 다이어그램을 생성하였을 때 각 복잡도 측정 기준에 따른 복잡도를 비교한 그래프이다. 연산의 유형과 상태 다이어그램의 복잡도를 함께 고려하면 상태의 수와 전이의 수가 함께 감소하며 복잡도가 감소한다.

연산의 유형과 상태 다이어그램의 복잡도를 함께 고려하면 상태 다이어그램의 복잡도를 고려한 경우와 마찬가지로 상태의 병합이 일어나 전체 상태의 수가 감소한다. 클래스 f의 경우, 전이의 복잡도가 감소하지 않아 상태 다이어그램 복잡도가 감소하지 않았다. 클래스 g의 경우 상태와 전이, 그리고 전이의 선행/후행 조건이 기본 생성 방식의 상태 다이어그램과 동일하여 복잡도가 감소하지 않았다. 그 외 7개 클래스에 대해 클래스에 대해 상태 다이어그램의 복잡도가 감소한다.

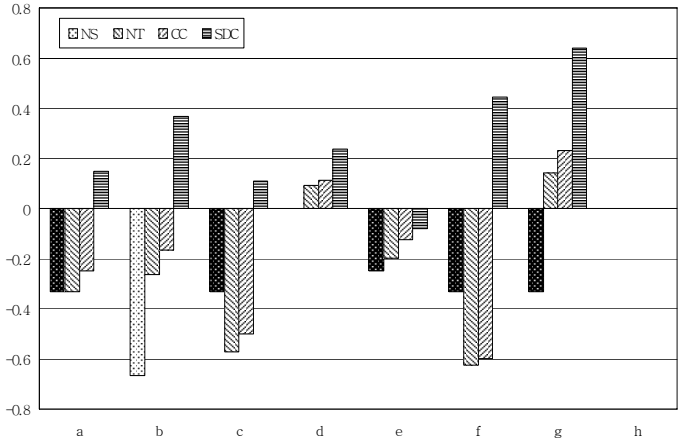
- 복잡도만 고려한 방식과 연산 유형 및 복잡도를 동시에 고려 방식의 비교

(그림 24)는 복잡도 고려 방식과 비교한 연산 유형 및 복잡도 고려 방식의 각 복잡도 측정 기준에 따른 복잡도를 비교한 그래프이다.

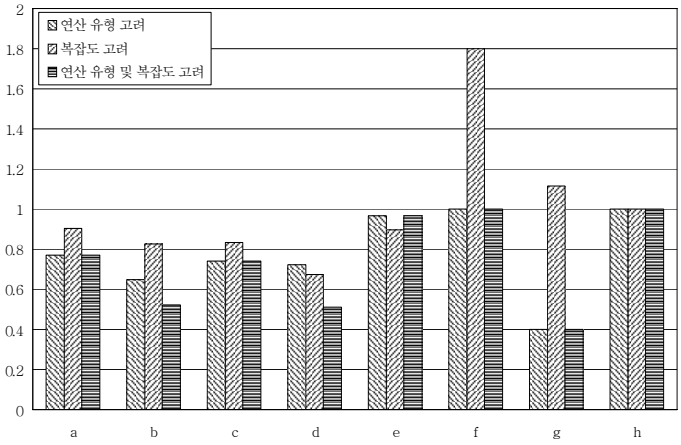
클래스 a, b, c, f에서 상태의 수와 전이의 수, CC값이 증가하였으나 SDC 복잡도가 떨어진 것을 확인할 수 있다. 이는 전이의 수가 증가하였지만 생성자와 소멸자 정보가 추가되어 전이의 선행/후행 조건이 간단해짐으로써, 전이의 복잡도가 감소한 것이다.



(그림 23) 연산 유형과 복잡도를 고려하여 생성한 상태 다이어그램의 복잡도 측정 기준별 복잡도



(그림 24) 복잡도 고려 방식과 비교한 연산 유형 및 복잡도 고려 방식의 각 복잡도 측정 기준에 따른 복잡도



(그림 25) 상태 다이어그램 생성 방식에 따른 상태 다이어그램 복잡도

클래스 e에서 상태와 전이의 수, CC값과 마찬가지로 SDC값 또한 증가한 것을 확인할 수 있다. 이는 기존의 복잡도 고려 방식만을 사용했을 경우, 8개의 액세서 연산이 뮤테이터 연산으로 생성되어서 생성 단계 및 합성 단계에서 연산의 복잡도를 감소시키지만, 연산 유형을 함께 고려하게 되면 액세서 연산의 복잡도를 고려하지 않은 채 상태 다이어그램을 생성하고 차후 액세서 연산을 추가하게 되어, 액세서 연산의 복잡도에 따라 최종 생성된 상태 다이어그램의 복잡도가 증가한 것을 의미한다. 이러한 경우, 액세서 연산은 상태 다이어그램의 상태를 변화시키지 않으며 수행 순서에도 영향을 미치지 않기 때문에, 상태 다이어그램의 복잡도와 연관이 없다고 할 수 있다.

• 다이어그램 생성 방식에 따른 SDC 복잡도 비교

(그림 25)은 상태 다이어그램 기본 생성 방법을 1로 하여 각 상태 다이어그램 생성 방법에 따른 상태 다이어그램의 SDC 복잡도를 비교한 그래프이다. 클래스 e를 제외한 나머지 7개의 클래스에서 연산의 유형과 복잡도를 모두 고려하여 생성한 상태 다이어그램의 SDC가 어느 하나만 고려한 상태 다이어그램의 SDC와 적어도 같거나 낮은 값을 가지는 것을 확인 할 수 있다. 그러므로 복잡도가 낮은 상태 다이어그램을 생성하기 위해서는 연산의 유형과 생성된 다이어그램의 복잡도를 모두 고려하여야 한다.

5. 관련 연구

기존에 시나리오로부터 동적 모델을 유도하는 연구는 많이 진행되었다[34]. 예를 들어 시나리오 집합으로부터 객체의 상태차트 다이어그램을 유도하려는 연구가 진행되었다[35-36]. 구체적으로 UML의 시퀀스 다이어그램으로부터 상태차트 다이어그램을 생성하는 연구가 진행되었다[37-38]. 통신 분야에서 많이 사용되는 MSC(Message Sequence Chart)는 시퀀스 다이어그램과 유사하며 이 MSC로부터 상태 다이어그램을 유도하려는 연구도 진행되었다[39].

이들 연구는 단편적으로 기술된 시스템의 기능을 조합/합성함으로써 전체적인 관점에서 시스템의 행위를 도출하고자 하는 연구이다. 가정하는 시나리오 또는 시퀀스 다이어그램은 단편적인 행위를 보여주기는 하지만 상태차트 다이어그램과 동일한 추상화 수준의 정보를 담고 있다. 즉 시스템의 구체적인 세부 행동을 메시지로서 정의하고 있다.

반면에 본 논문에서 입력으로 가정하는 클래스의 정적 모델은 각 연산의 선행/후행 조건과 연산의 유형이 기술된 것을 가정하며, 연산 내부의 동작에 대한 정보를 이용하지는 않는다. 따라서 위에서 언급된 많은 연구 기법들을 적용하기 어렵다.

앞에서 언급되었듯이 테스팅, 정형적 분석, 코드 생성 등의 연구 문헌에서 선행/후행 조건으로부터 시스템/클래스의 동적 모델 자동 생성에 대한 필요성이 제기되고 있다[17-19]. 그러나 선행/후행 조건으로부터 자동 생성되는 동적 모델은 복잡도가 높으므로, 생성과 분석에 들이는 노력에 비해 얻을 수 있는 정보가 적다. 실제 선행/후행 조건으로부터 동적 모델을 자동 생성하는 연구는 활발히 이루어지지 않고 있다.

Nebut과 Fleurey는 선행/후행 조건이 추가된 유스케이스로부터 유스케이스 간의 동적 모델을 표현하는 유스케이스 전이 시스템(Usecase Transition System)을 구축하는 방안으로서 시뮬레이션을 언급하고 있다[17]. 즉 유스케이스의 실제 수행 순서가 주어진 선행/후행 조건으로부터 결정되기 위해서는 선행/후행 조건에서 사용하는 각종 매개변수의 값이 필요하며 이 논문에서는 임의의 매개변수의 값을 무수히 시도함으로써 유스케이스의 수행 순서를 결정하여 유스케이스 전이 시스템을 구축한다. 이 논문에서 언급하는 전체 시뮬레이션(Exhaustive Simulation)은 마치 프로그램 테스팅과 유사한 개념으로서 선택된 매개변수 값에 따라서 결과가 달라질 수 있으므로 정확한 유스케이스 전이 시스템을 구축한다는 보장이 불가능하다. Riebisch와 Philippow도 선행/후행 조건이 있는 유스케이스로부터 동적 모델을 구성하는 방법을 언급하고 있지만[19], 구체적인 알고리즘으로서 자동화가 되기 불가능한 수준으로만 설명하고 있다.

Dick과 Faivre은 각 연산에 주어진 선행/후행 조건으로부

터 DFA(Deterministic Finite Automata)를 구성하여 테스트 시나리오를 도출하려고 하였다[18]. 이 연구에서는 주어진 연산의 선행/후행 조건에 대해서 분할 분석(Partition Analysis)을 적용하여 각 연산을 세부화시킨 후에 세분화된 연산을 바탕으로 한 DFA 구축 방법을 소개하고 있다. 이 연구가 선행/후행 조건을 이용하고 자동적인 구축 방법을 제시한다는 측면에서는 많은 관련이 있지만 구축된 DFA가 초기에 주어진 연산들이 아니라 세부화된 즉 분할된(partitioned) 연산들이기 때문에 이 방법을 클래스의 동적 모델에 적용하기는 어렵다. 다시 말하면 이 방법에 따르면 클래스의 연산1이 연산1-1, 연산1-2 등과 같이 분할되어 전이에 표시되므로 클래스의 동적 모델이라고 간주하기 어렵다.

6. 결론 및 향후 연구 방향

상태 다이어그램은 테스팅이나 정형적 검증등의 다양한 분야에서 유용하게 사용할 수 있는 동적 모델이다. 그러나 상태 다이어그램을 생성하고, 적합한지 검증하는 과정은 복잡하다. 그에 비해, 클래스 연산의 선행/후행 조건을 정의하는 것은 간단하다. 클래스 연산의 선행/후행 조건을 통해 상태 다이어그램을 자동으로 생성하는 연구는 아직 활발히 이루어지지 않았다.

본 논문에서는 클래스 연산의 선행/후행 조건을 이용하여 여러 응용 분야에서 자동으로 분석하고 사용하기 용이한 상태 다이어그램을 자동으로 생성하는 방법을 제시하였다. 상태 다이어그램을 자동으로 생성하기 위해, 연산의 선행/후행 조건의 포함 관계를 이용하여 단위 상태 다이어그램을 생성하는 방법을 제시하였고, 단위 상태 다이어그램의 상태 사이의 포함 관계를 이용하여 복수 개의 단위 상태 다이어그램을 합성하는 방법을 제시하였다.

또한 생성된 최종 상태 다이어그램의 복잡도를 감소시키기 위해 연산의 종류를 고려한 상태 다이어그램 생성 방법을 제안하고, 상태 다이어그램의 복잡도를 측정하는 매트릭을 정의하였다. 제시된 방법을 토대로 상태 다이어그램을 자동으로 생성하는 도구인 SDAG를 제작하였고, 단위 상태 다이어그램 생성과 합성 단계에서 연산의 종류를 고려하고 복잡도 메트릭을 사용하여 더 낮은 복잡도를 가지는 상태 다이어그램을 생성할 수 있도록 하였다.

현재 구현된 상태 다이어그램 자동 생성 도구는 몇 가지 개선사항이 존재한다. 먼저, 사용자 가독성을 높이기 위한 상태 다이어그램 생성 방법이 연구되어야 한다. 현재 SDAG는 자동 처리에 적합한 상태 다이어그램을 생성한다. 하지만 클래스의 모든 정보를 통해 생성되지 않는 한계가 있으므로, SDAG에 의해 생성되는 상태 다이어그램은 일반적으로 사용자가 기대하는 상태 다이어그램보다 더 복잡할 수 있다. 따라서 자동 분석을 하기 전에 사용자에 의해 수정을 거쳐야 할 필요가 있다. 이를 위하여 사용자 가독성을 고려한 상태 다이어그램을 생성하도록 개선할 필요가 있다. 아울러 자동으로 분석된 상태의 조건 또한 사용자가 알아보기 쉽도록 수정할 필요가 있다.

또한 자동 분석 도구에 근거한 복잡도 메트릭 연구도 필요하다. SDAG는 복잡도를 측정하기 위해 전이의 복잡도와 상태의 복잡도를 함께 고려하였다. 그러나 이 복잡도는 자동 분석 도구의 실제 분석 방식에 근거하여 만들어진 것이 아닌, 개념적으로 만들어진 것으로, 실제 자동 분석 도구의 처리 속도를 증가시키기 위해 마련된 것이 아니다. 따라서 자동 분석 도구에 적용함과 함께, 자동 분석 도구의 분석 방식에 근거하여 더욱 정확한 복잡도 메트릭을 만들 필요가 있다.

참 고 문 헌

[1] T. S. Chow. Testing software design modeled by finite-state machines. IEEE Trans. Software Eng., 4(3):178-187, 1978.

[2] P. Chevalley and P. Th?venod-Fosse. Automated generation of statistical test cases from uml state diagrams. In COMPSAC, pp.205-214, 2001.

[3] H. S. Hong, Y. G. Kim, S. D. Cha, D. -H. Bae, and H. Ural. A test sequence selection method for statecharts. Softw. Test., Verif. Reliab., 10(4):203-227, 2000.

[4] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from uml state diagrams.

[5] G. Antoniol, L. C. Briand, M. D. Penta, and Y. Labiche. A case study using the round-trip strategy for state-based class testing. In ISSRE, pp.269-279, 2003.

[6] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on state-chart. In ICSE, pp.86-95, 2004.

[7] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. IEEE Trans. Software Eng., 30(11):770-793, 2004.

[8] A. David, M. O. M?ller, and W. Yi. Formal verification of uml statecharts with real-time extensions. In FASE, pp.218-232, 2002.

[9] J. van Katwijk, H. Toetenel, A.-E.-K. Sahraoui, E. Anderson, and J. Zalewski. Specification and verification of a safety shell with statecharts and extended timed graphs. In SAFECOMP, pp.37-52, 2000.

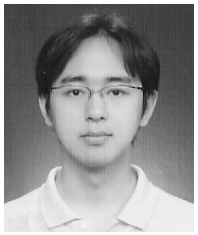
[10] K. C. Kang and K.-I. Ko. Formalization and verification of safety properties of statechart specifications. In APSEC, pp.16-27, 1996.

[11] G. Pint?r and I. Majzik. Runtime verification of statechart implementations. In WADS, pp.148-172, 2004.

[12] H. J. K?hler, U. Nickel, J. Niere, and A. Z?ndorf. Integrating UML diagrams for production control systems. In ICSE '00: Proceedings of the 22nd international conference on Software engineeering, pp.241-251, New York, NY, USA, 2000. ACM Press.

[13] I. Niaz and J. Tanaka. Code generation from UML statecharts. In SEA '03: Proceedings of international conference on

Software Engineering and Applications, 2003.
[14] R. Knor, G. Trausmuth, and J. Weidl. Reengineering c/c++ source code by transforming state machines. In ESPRIT ARES Workshop, pp.97-105, 1998.
[15] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In ICSE, pp.439-448, 2000.
[16] N. Pywes and P. Rehmet. Recovery of software design, state-machines, and specifications from source code. In ICECCS, pp.279-288, 1996.
[17] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. J?z?quel. Automatic test generation: A use case driven approach. IEEE Trans. Software Eng., 32(3):140-155, 2006.
[18] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In FME, pp.268-284, 1993.
[19] M. Riebisch, I. Philippow, and M. G?tze. UML-based statistical test case generation. In NetObject-Days, pp.394-411, 2002.
[20] B. Meyer. Applying "Design by Contract", In Computer, 25 (10):40-51, 1992.
[21] OMG. UML 2.1 Superstructure specification. "http://www.uml.org".
[22] J. Warmer and A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, 2nd edition, 2003.
[23] R. Kramer. iContract - the java(tm) design by contract(tm) tool. In TOOLS (26), pp.295-307, 1988.
[24] M. Karaorman and P. Abercrombie. jContractor: Introducing design-by-contract to java using reflective bytecode instrumentation. Formal Methods in System Design, 27(3):275-312, 2005.
[25] J. Newmarch. Adding contracts to java. In TOOLS (27), pp. 2-7, 1988.
[26] S. H. Edwards, M. Sitaraman, B. W. Weide, and J. E. Hollingsworth. Contract-checking wrappers for c++ classes. IEEE Trans. Software Eng., 30(11):794-810, 2004.
[27] P. Guerreiro. Simple support for design by contract in c++. In TOOLS (39), pp.24-34, 2001.
[28] R. Pl?sch and J. Pichler. Contracts: From analysis to c++ implementation. In TOOLS (30), pp.248-257, 1999.
[29] McCabe, T. A Complexity Measure. IEEE Trans. Software Eng., 2(4):308-320, 1976.
[30] Cruz-Lemus, J. A., Genero, M. Olivas, J. A., Romero, F.P. Piattini, M. Predicting UML statechart diagrams understandability using fuzzy logic-based techniques. Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering SEKE 2004, pp.238-245, 2004.
[31] Stump, A., Barrett, C.W., Dill., D.L.: CVC: a cooperating validity checker. In Proc. of CAV'02. Vol.2404 of LNCS. 2002.
[32] E. Koutsofios and S. C. North. Drawing graphs with dot. AT&T Bell Laboratories, Murray Hill, NJ.
[33] JML. The Java Modeling Language. "http://www.eecs.ucf.edu/~leavens/JML/",
[34] H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In SCESM, pp.5-12, 2006.
[35] J.Whittle and J. Schumann. Generating statechart designs from scenarios. In ICSE, pp.314-323, 2000.
[36] J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: An air traffic control case study. Software and System Modeling, 4(1):71-93, 2005.
[37] E. M?kinen and T. Syst?. Mas - an interactive synthesizer to support behavioral modeling in uml. In ICSE, pp.15-24, 2001.
[38] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In ICSE, pp.188-197, 2001.
[39] M. Mukund, K. N. Kumar, and M. A. Sohoni. Synthesizing distributed finite-state systems from mscs. In CONCUR, pp. 521-535, 2000.



이 광 민

e-mail : leekm@pusan.ac.kr
2005년 부산대학교 전자전기정보컴퓨터공학부 정보컴퓨터공학전공(학사)
2008년 부산대학교 컴퓨터공학과(공학석사)
2008년~현 재 (주)사이버맵월드부설연구소 근무
관심분야 : 소프트웨어 공학, 알고리즘, 자동처리, 오토마타 등

배 정 호

e-mail : jhbae83@pusan.ac.kr
2007년 부산대학교 전자전기정보컴퓨터공학부 정보컴퓨터공학전공(학사)
2009년 부산대학교 컴퓨터공학과(공학석사)
2009년~현 재 부산대학교 컴퓨터공학과 박사과정
관심분야 : 소프트웨어공학, TDD, 리팩토링, 디자인 패턴 등

채 흥 석

e-mail : hschae@pusan.ac.kr
1994년 서울대학교 원자핵공학(학사)
1996년 한국과학기술원 전산학(석사)
2000년 한국과학기술원 전산학(박사)
2000년~2003년 ㈜ 동양시스템즈 기술연구소 선임연구원
2003년~2004년 한국과학기술원 전산학과 초빙교수
2004년~현 재 부산대학교 컴퓨터공학과 조교수
관심분야 : 객체지향 방법론, 소프트웨어 테스팅, 소프트웨어 메트릭, 소프트웨어 유지보수, 미들웨어 설계, 프로덕트라인 공학