Java 다중 스레드 프로그램을 위한 오토마타 기반 테스팅 환경의 설계 및 구현

(The Design and Implementation of Automata-based Testing Environments for Multi-thread Java Programs)

서 희 석 † 정 인 상 †† 김 병 만 ††† 권 용 래 ††††

(Heui-Seok Seo) (In Sang Chung) (Byeong Man Kim) (Yong Rae Kwon)

요 약 고전적인 결정적 테스팅 방법은 명세와 프로그램의 동치 관계를 기반으로 병행 프로그램의 수행 경로를 제어한다. 따라서, 주어진 시퀀스를 직접 구현하지 않고, 그와 의미적으로 동일한 다른 시퀀스를 구현한 프로그램에 대해서는 결정적 테스팅 방법을 적용하기 어렵다. 이를 해결하기 위해서, 우리는 테스트 시퀀스와 의미적으로 동일한 모든 시퀀스들을 허용하는 동치 집합 오토마타를 이용한 오토마타 기반 테스팅 방법을 제안하였었다. 이 논문에서는 Java 다중 스레드 프로그램에 대한 오토마타 기반 테스팅 환경을 제안하고, 테스팅 환경 내의 테스트 수행 지원 도구를 설계하고 구현하는 방법을 제안한다. 테스트 수행 지원 도구에서는 주어진 Java 다중 스레드 프로그램을 오토마타 기반의 결정적 테스팅 방법이 적용된 프로그램으로 변환하고, 이 변환된 프로그램을 수행함으로써 테스트의 결과를 알 수 있다. 이를 위해서 테스트 수행 지원 도구 내에서 동치 집합 오토마타를 생성하는 오토마타 생성기와 프로그램의 수행을 제어하기 위한 재연 제어기를 설계하고 구현한다. 그리고, 가스 충전소 예제를 이용하여 오토마타 기반의 결정적 테스팅의 과정 및 효과를 기술한다.

키워드 : 자바 다중 스레드 프로그램, 테스팅 환경, 재연 제어기, 동치 집합 오토마타, 결정적 시험

Abstract Classical deterministic testing controls the execution of concurrent programs based on the equivalence between specifications and programs. However, it is not directly applicable to a situation in which synchronization sequences, being valid but infeasible, are taken into account. To resolve this problem, we had proposed automata-based deterministic testing in our previous works, where a concurrent program is executed according to one of the sequences accepted by the automaton recognizing all sequences semantically equivalent to a given sequence. In this paper, we present the automata-based testing environment for Java multi-thread programs, and we design and implement "Deterministic Executor" in the testing environment. "Deterministic Executor" transforms a Java multi-thread program by applying automata-based deterministic testing, the transformed program presents testing results. "Deterministic Executor" uses "Automata Generator", which generates an equivalent automaton of a test sequence, and "Replay Controller", which controls the execution of programs according to the sequence accepted by the automaton. By illustrating automata-based testing procedures with a gas station example, we show how the proposed approach does works in a Java multi-threaded program.

Key words : Java Multi-thread Program, Testing Environments, Replay Controller, Automata for Equivalent Class, Deterministic Testing

· 이 논문은 정보통신연구진흥원에서 지원하는 2001년 대학기초연구지원사업(과제번호 : 2001-142-2)에 의해서 일부 지원되었음.

† 비 회 원 : 한국과학기술원 전자전산학과
hsseo@salmosa.kaist.ac.kr

†† 종신회원 : 한성대학교 컴퓨터 공학부 교수
insang@hansung.ac.kr

††† 종신회원 : 금오공과대학교 교수
bmkim@cespc1.kumoh.ac.kr

†††† 종신회원 : 한국과학기술원 전자전산학과 교수
kwon@salmosa.kaist.ac.kr

논문접수 : 2002년 5월 8일
심사완료 : 2002년 9월 30일

1. 서 론

Java 프로그램에서 다중 스레드 기법은 프로그램의 전체적인 성능을 향상시키고 주어진 자원의 활용도를 높일 수 있다. 그러나, 일반적인 병행 프로그램과 마찬가지로 Java 다중 스레드 프로그램은 비결정성을 내포하고 있기 때문에 동일한 입력 자료를 반복 수행했을 때 서로 다른 결과를 유발할 수 있다. 이와 같은 비결정적인 특징으로 인하여 Java 다중 스레드 프로그램과 같은 병행 프로그램의 테스팅 및 검증은 여러 가지 문제점들을 가진다 [1, 2, 3, 4]. 먼저 스레드들 간의 병행성으로 인하여 동일한 입력 자료에 대해서 하나의 수행 경로가 옳은 결과를 산출했더라도 다른 수행 경로에서는 잘못된 결과를 산출할 수 있기 때문에, 주어진 입력 자료만으로는 프로그램의 오류 여부를 판단하기 어렵다. 또한 입력에 대한 수행 경로가 프로그램 내에서 비결정적으로 선택되기 때문에, 어느 수행 경로에 에러가 위치하는지를 확인하기가 어려워진다. 이러한 비결정성에 의한 테스팅의 어려움을 해결하기 위해서, 병행 프로그램의 수행 경로를 제어하여 테스팅을 수행하는 결정적 수행 테스팅(deterministic execution testing) 방법이 개발되었다[5, 6, 7, 8, 9].

결정적 테스팅(결정적 수행 테스팅)에 대한 선행 연구들은 명세와 프로그램 사이의 동치 관계를 기반으로 하고 있다. 즉, 명세에서 기술한 모든 동기화 시퀀스들에 대해서 명세적으로 타당한(valid) 시퀀스가 실행되지 않거나(infeasible), 타당하지 않은(invalid) 시퀀스가 실행되는(feasible) 경우에 병행 프로그램이 에러를 가지는 것으로 정의하며, 명세적으로 타당한 시퀀스들을 테스트 데이타로 이용하여 결정적 테스팅을 수행한다 [10, 11, 12]. 그러나, 병행 프로그램에서는 설계 단계에서의 결정에 의하여 명세적으로 타당한 시퀀스들 중에서 일부만이 구현되는 경우를 허용하며, 이러한 경우에 명세와 프로그램의 동치 관계를 기반으로 하는 기존의 결정적 테스팅을 직접적으로 적용하기가 어렵다. 예를 들면, 전화를 거는 과정에서 호출자가 송신음을 듣는 이벤트와 피호출자가 수신음을 듣는 이벤트의 순서는 비결정적이다. 따라서 호출자가 송신음을 들은 후에 피호출자가 수신음을 듣는 시퀀스와 그 반대의 순서를 가지는 시퀀스가 있을 수 있으며, 이 중에서 어느 한 시퀀스만 구현되더라도 시스템에는 오류가 없다. 이와 같은 경우에 명세적으로 타당한 모든 시퀀스들이 실행되지 않음에도 불구하고 프로그램은 명세를 만족한다. 그러나, 기존의 결정적 테스팅에서는 이러한 상황을 확인할 수 있는 적절한 방법을 제공하지 않기 때문에, 우리는 주어진 시퀀스에 대해서 프로그램 행위에 동일한 효과를 주는 다른 실행 가능한 시퀀스가 있는지를 조사할 수 있는 오토마타 기반의 새로운 결정적 테스팅 방법을 제안하였다[13, 14, 15].

오토마타 기반의 결정적 테스팅에서는 병행 프로그램에서의 동기화 오류를 다음과 같이 정의하였다.

- $[\omega]_{(E,D)}$를 명세 S의 입력 x에 대한 동기화 시퀀스 ω와 동일한 효과를 갖는 유효한 동기화 시퀀스들의 집합이라 하고, $feasible(P, x)$를 입력 x에 대한 프로그램 P에서 실행 가능한 동기화 시퀀스들의 집합이라고 하자. 그러면, 입력 x에 대해 $[\omega]_{(E,D)} \cap feasible(P, x) = \Phi$일 때 프로그램 P는 동기화 오류를 갖는다고 말한다.

이러한 동기화 오류를 테스팅하기 위한 오토마타 기반의 결정적 테스팅 방법은 다음과 같은 순서로 진행된다. 먼저 주어진 동기화 시퀀스 w에 대해서, 동기화 이벤트들(E) 사이의 의존 관계(D)를 이용하여 w에 속하는 이벤트들의 부분 순서(partial-order)를 표현하는 선후 관계 그래프($P_{[\omega, D]}$)를 생성한다. 다음으로 우선 순위 그래프에서 독립적이고 병행적인 이벤트들을 인터리빙 하여 w에 대한 동치 집합 오토마타를 생성한다. 생성된 동치 집합 오토마타는 $[\omega]_{(E,D)}$에 속하는 모든 동기화 시퀀스를 받아들인다. 마지막으로 동치 집합 오토마타에 의해서 받아들여지는 동기화 시퀀스들 중에서 하나라도 프로그램에서 실행되는 경우에는 그 프로그램에 동기화 오류가 없는 것으로 판단한다. 따라서, 기존의 결정적 테스팅 방법에 비해서 오토마타 기반의 결정적 테스팅 방법에서는 주어진 시퀀스와 동일한 효과를 보이는 추가적인 시퀀스들을 고려한다.

이 논문에서는 Java 다중 스레드 프로그램을 대상으로 하는 오토마타 기반 테스팅 환경을 제안하며, 테스트 환경에서 테스트 수행을 지원하기 위한 도구의 설계 및 구현 방법에 대해서 제안한다. 테스트 수행 지원 도구에서는 테스트 대상이 되는 Java 다중 스레드 프로그램을 오토마타 기반의 결정적 테스팅 방법이 적용된 프로그램으로 변환하고, 변환된 프로그램을 수행함으로써 테스트 결과를 알 수 있다. 이를 위하여 테스트 수행 지원 도구에서는 *오토마타 생성기*와 *재연 제어기(replay controller)*를 이용한다. *오토마타 생성기*는 이벤트들의 의존 관계를 이용하여 주어진 테스트 시퀀스에 대한 동치 집합 오토마타를 생성하며, *재연 제어기*는 동치 집합 오토마타를 기반으로 하여 Java 다중 스레드 프로그램

의 수행을 제어한다. 이 논문에서는 가스 충전소 예제를 이용하여 각 도구들의 기능을 기술한다.

이 논문의 구성은 다음과 같다. 2장에서는 명세와 프로그램의 동치 관계를 기반으로 하는 고전적인 결정적 테스팅 방법에 대해서 간략히 설명한다. 3장에서는 Java 다중 스레드 프로그램에 적용할 수 있는 오토마타 기반 테스팅 환경을 기술한다. 4장에서는 주어진 시퀀스에 대한 동치 집합 오토마타를 생성하는 *오토마타 생성기*와 오토마타에 따라서 프로그램의 수행을 제어하기 위한 *재연 제어기*를 설계하고 구현하는 방법에 대해서 기술하며, 오토마타 기반의 결정적 테스팅 방법을 적용하여 Java 다중 스레드 프로그램을 변환하는 방법에 대해서 설명한다. 5장에서는 구현된 오토마타 기반 테스팅 환경의 테스트 수행 지원 도구에 대해서 기술하고, 이를 이용하여 오토마타 기반의 결정적 테스팅의 효과를 설명한다. 마지막으로 6장에서는 결론 및 향후 연구 방향을 기술한다.

2. 고전적인 결정적 테스팅

병행 프로그램의 재수행성을 보장하기 위한 방법인 고전적인 결정적 테스팅 방법은, 일반적으로 동기화 시퀀스를 선택하는 과정과 선택된 시퀀스를 강제적으로 수행하는 과정으로 구성된다. 먼저 동기화 시퀀스를 수집하고 선택하는 방법으로, 프로그램의 수행 시에 이벤트 발생 순서 및 관련 정보를 기록하는 방법이 있으며[6, 7], 보다 체계적인 방법으로는 명세나 프로그램의 구조, 원시 코드 등을 분석하여 동기화 시퀀스를 얻는 방법이 있다[1, 8, 11, 12]. 동기화 시퀀스를 선택한 후에는 선택된 시퀀스와 일치하도록 프로그램의 수행 경로를 강제하는 방법이 필요하다. 이를 위한 전형적인 방법으로는 제어기 프로세스(controller process, 재연 제어기)를 도입하여 프로그램의 수행 경로를 강제하는 방법이 있다[3, 4, 5, 7, 8, 9]. 이 방법에서는 테스트 대상이 되는 프로그램을 재연 제어기와 통신할 수 있도록 변경하며, 변경된 프로그램에서 동기화 이벤트를 실행할 때마다 재연 제어기는 테스트 시퀀스를 기준으로 이벤트의 실행 가능 여부를 판단한다. 이 때, 재연 제어기는 실행이 허용되지 않은 이벤트의 실행을 보류하며, 다른 이벤트들이 실행된 후에 보류된 이벤트의 실행 여부를 다시 판단하는 방법으로 프로그램의 수행 경로를 제어한다.

정확한 설명을 위해서, 생산자 소비자 문제를 구현한 Java 다중 스레드 프로그램의 수행 경로를 강제하는 방법을 설명한다(그림 1). 먼저 그림 1(A)의 생산자/소비자 프로그램에서 병행적인 행위를 하는 스레드로 *Producer*와 *Consumer*가 있으며, *Producer*에서는 아이템을 생성하여 버퍼에 저장하고(*put(X)*), *Consumer*에서는 버퍼에 저장된 아이템을 사용한다(*get()*). 그림 1(B)은 프로그램의 결정적 수행을 위해서 변환된 프로그램의 구조이다. 변환되는 부분은 두 가지로, 하나는 이벤트의 수행 허용 여부를 판단하는 재연 제어기(*Replay Controller*)가 프로그램에 추가되며, 다른 하나는 각 스레드의 메소드들 전후에 재연 제어기의 메소드인 *reqPermit(tID)*/*done(tID)*의 호출이 추가된다. *reqPermit(tID)* 메소드는 주어진 시퀀스를 기준으로 하여 스레드에서 요구하는 메소드가 수행 가능한지를 판단하고, 수행 가능한 이벤트의 경우에는 수행을 허용하지만 그렇지 않은 경우에는 스레드의 수행을 지연(block)시킨다. *done(tID)* 메소드는 스레드의 메소드 실행이 끝났음을 재연 제어기에 알리고, 지연되었던 스레드들을 모두 활성화시킨다.

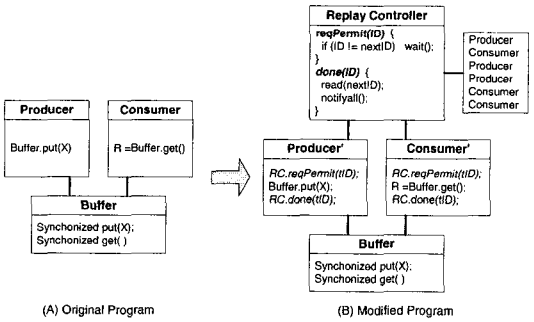

그림 1 (A) 원래의 생산자/소비자 프로그램의 구조와 (B) 결정적 테스팅 수행 방법을 적용하여 변환된 프로그램의 구조

그림 1(B)의 프로그램 수행이 제어되는 과정을 보면, 프로그램에서 첫 번째 수행 예정인 스레드는 *Producer*이다. 그런데, 프로그램에서 수행을 요청한 스레드가 *Customer*인 경우에는 그 실행이 지연된다. 다음으로 *Producer* 스레드가 수행을 요청한 경우에는 *put(X)* 메소드를 실행하며, 새로운 수행 예정인 스레드는 *Customer*가 된다. *Producer* 스레드에서의 실행이 끝난 후에는 지연되었던 *Customer* 스레드가 활성화되며, 이는 재연 제어기에 의해서 수행이 허용된다. 이와 같은 방법으로 생산자/소비자 프로그램은 스레드의 실행 요청 순서와는 무관하게 항상 주어진 시퀀스에 따른 수행 경로를 따르도록 제어된다. 한편, 테스팅을 수행하는 중에

한 스레드가 일정 이상의 시간 동안 수행이 허용되지 않은 경우에는, 주어진 시퀀스가 그 프로그램에서 수행 불가능하다는(infeasible) 결론을 내린다.

이와 같이 고전적인 결정적 테스팅에서는 주어진 동기화 시퀀스에 따라서 프로그램의 수행 순서를 강제화하여 프로그램을 실행시키고, 주어진 시퀀스가 실행 가능한지를 판단한다. 그러나, 주어진 시퀀스가 실행 불가능하다고 판단한 경우에 그와 동일한 행위를 하는 다른 시퀀스가 존재하는지를 확인하는 등의 다른 조치를 취하지 않는다. 이러한 점을 보완하기 위해서, 오토마타 기반의 결정적 테스팅에서는 주어진 시퀀스의 동치집합을 받아들이는 오토마타를 이용하여, 주어진 시퀀스뿐만 아니라 그와 동일한 행위를 하는 모든 시퀀스에 대해서 프로그램을 테스트한다[13, 14, 15].

3. 오토마타 기반 테스팅 환경

고전적인 결정적 테스팅 방법들이 주어진 시퀀스만을 근거로 하여 프로그램의 실행 가능성을 판단하는데 반해서, 오토마타 기반의 결정적 테스팅에서는 주어진 시퀀스의 동치 집합 오토마타를 이용하여 프로그램의 실행 가능성을 판단한다. 따라서, 오토마타 기반의 결정적 테스팅을 병행 프로그램에 적용하기 위해서는 새로운 테스팅 환경이 필요하다. 이 절에서는 Java 다중 스레드 프로그램을 위한 오토마타 기반의 테스팅 환경을 제안한다. 오토마타 기반 테스팅 환경에서는 명세와 Java 다중 스레드 프로그램을 입력으로 받아서 프로그램에 오토마타 기반의 결정적 테스팅을 적용한 테스트 결과를 출력한다. 이와 같은 작업을 하는 테스트 환경은 테스트 데이타를 생성하는 부분(front end)과 테스트를 수행하는 부분(back end)으로 구성된다 (그림 2). 테스트 데이타 생성 부분에서는 주어진 명세를 분석하여 테스트 시퀀스와 이벤트들의 의존성 테이블을 생성하며, 테스트 수행 부분에서는 이들 정보를 Java 다중 스레드 프로그램에 적용하여 오토마타 기반의 결정적 테스팅을 수행한다. 이 논문에서는 테스트 수행 부분의 설계 및 구현에 대해서 기술한다.

테스트 수행 부분에서는 테스트 시퀀스, 의존성 테이블, Java 다중 스레드 프로그램을 입력으로 받아서, Java 다중 스레드 프로그램을 오토마타 기반의 결정적 테스팅을 적용한 프로그램으로 변환하고, 변환된(transformed) 프로그램을 실행함으로써 테스팅의 결과를 얻는다. 변환된 프로그램은 테스트 시퀀스에 대한 동치 집합 오토마타, 수행을 제어하기 위해서 코드를 삽입한 변경된(modified) Java 프로그램, 그리고 Java 다중 스레

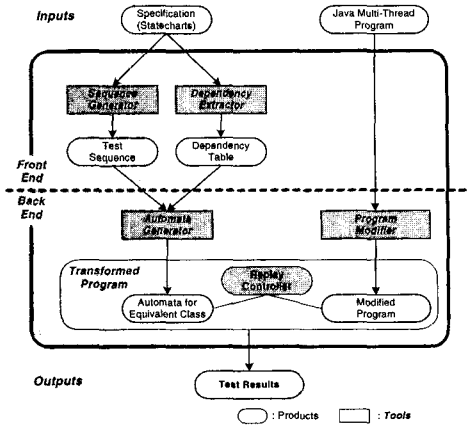

그림 2 Java 다중 스레드 프로그램에 대한 오토마타 기반 테스팅 환경

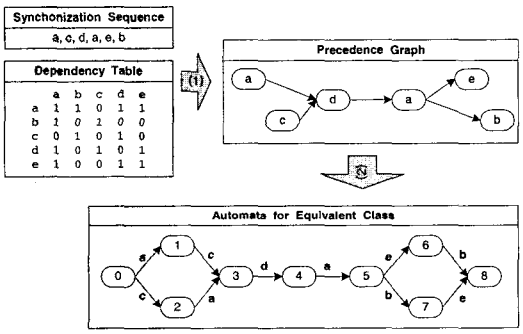

그림 3 의존성 테이블을 이용한 테스트 시퀀스의 동치 집합 오토마타 생성 과정

드 프로그램을 위한 재연 제어기로 구성된다. 이를 위해서, 테스트 수행부에서는 오토마타 생성 도구와 프로그램 변경 도구가 필요하다. **오토마타 생성기**는 이벤트들의 의존성 테이블을 이용하여 (1) 테스트 시퀀스에 대한 선후 관계 그래프를 생성한 다음 (2) 선후 관계 그래프로부터 동치 집합 오토마타를 생성한다(그림 3). 선후 관계 그래프는 테스트 시퀀스의 이벤트들이 반드시 만족해야 하는 부분 순서들을 표현하며, 동치 집합 오토마타에서는 테스트 시퀀스와 동일한 행위를 수행하는 모든 시퀀스들을 받아들인다. 한편, **프로그램 변경기**는 Java 다중 스레드 프로그램의 비결정적인 수행 순서를 제어하기 위해서 재연 제어기의 메소드를 호출하는 코드를 삽입한다. 이 도구들의 결과물과 함께 변환된 프로그램을 구성하는 **재연 제어기**는 고전적인 결정적 테스팅에서의 재연 제어기와 유사한 구조를 가지지만, 오토

마타를 기준으로 프로그램의 수행을 제어할 수 있도록 다시 설계된다. 따라서, 이 논문에서 설계되고 구현된 재연 제어기는 모든 Java 다중 스레드 프로그램들에 적용할 수 있다.

4. 오토마타 기반 테스팅 과정

이 절에서는 오토마타 기반 테스팅 환경의 테스트 수행 부분을 담당하는 오토마타 생성기와 재연 제어기의 설계 및 구현 방법에 대해서 설명한다. 또한, 이들을 이용하여 오토마타 기반의 결정적 테스팅 방법이 적용된 새로운 Java 프로그램을 생성하는 방법 및 과정을 기술한다. 오토마타 기반 테스팅의 적용 과정에서 나타나는 결과는 가스 충전소 (gas station) 예제를 이용하여 설명한다.

4.1 가스 충전소 예제

병행 프로그램의 예제로 자주 사용되는 가스 충전소 예제는 운영자와 펌프, 손님의 3가지 요소로 구성되며 [11, 16], 한정된 운영자와 펌프를 이용하여 다수의 손님에게 병행적으로 서비스를 제공해야 하는 특징을 가진다 (그림 4). 가스 충전소 예제에서 발생하는 행위의 순서를 보면, 주유소에 도착한 손님은 운영자에게 자신이 사용할 펌프에 대한 요금을 지불하고(*prepay()*), 운영자는 그 펌프를 활성화 시켜서 손님이 사용할 수 있도록 한다(*activate()*).

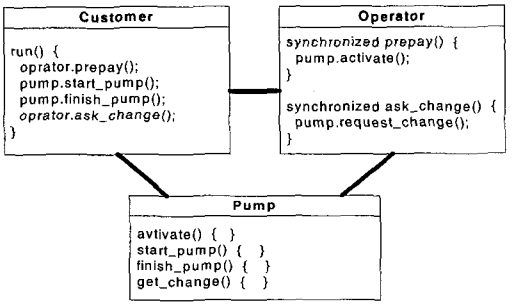

그림 4 가스 충전소 예제의 구조: 다수의 Customer가 병행적인 스레드들이다.

만약 펌프가 다른 손님에 의해서 사용중일 때는 순서를 기다리게 된다. 펌프를 배정 받은 손님은 가스를 충전하고(*start_pumping()*, *finish_pumping()*), 운영자에게 잔돈을 요구한다(*ask_change()*). 이에 대해서 운영자는 펌프에서 가스 사용량을 확인하여 잔돈을 거슬러준다 (*get_change()*).

Java 다중 스레드 프로그램에서 동기화 이벤트는 가스 충전소 예제의 손님 스레드와 같은 병행적인 스레드들에서 사용되는 이벤트(메소드)들이다. 이와 같은 동기화 이벤트의 수행 순서는 본질적으로 비결정적이지만 명세에서 요구하는 이벤트들 간의 의존성에 의해서 부분 순서가 존재한다. 가스 충전소 예제에서 동기화 이벤트 E_{ijk}는 손님 i가 펌프 j에 대해서 이벤트 k의 기능을 요구하는 경우이다. 이 때, 이벤트 k는 손님이 요구하는 4가지 이벤트 중의 하나를 나타낸다 - 1: *prepay()*, 2: *start_pumping()*, 3: *finish_pumping()*, 4: *ask_change()*. 가스 충전소 예제에서 같은 손님이 요구하는 동기화 이벤트들 사이에 의존성이 있으며, 같은 펌프를 사용하는 동기화 이벤트들 사이에도 의존성이 있다. 또한, 같은 운영자와 관련되는 동기화 이벤트들 사이에도 의존성이 존재한다. 즉, 다음과 같은 경우에 두 이벤트 E_{ijk} 와 E_{pqr}는 의존성을 가진다.

$$(E_{ijk}, E_{pqr}) \leftrightarrow (i = p) \text{ or } (j = q) \text{ or } ((k=1 \text{ or } 4) \text{ and } (r=1 \text{ or } 4))$$

표 1 주유소 문제에 대한 의존성 테이블

Event	E_{001}	E_{002}	E_{003}	E_{004}	E_{111}	E_{112}	E_{113}	E_{114}	E_{201}	E_{202}	E_{203}	E_{204}
E_{001}	1	1	1	1	1	0	0	1	1	1	1	1
E_{002}	1	1	1	1	0	0	0	0	1	1	1	1
E_{003}	1	1	1	1	0	0	0	0	1	1	1	1
E_{004}	1	1	1	1	1	0	0	1	1	1	1	1
E_{111}	1	0	0	1	1	1	1	1	1	0	0	1
E_{112}	0	0	0	0	1	1	1	1	0	0	0	0
E_{113}	0	0	0	0	1	1	1	1	0	0	0	0
E_{114}	1	0	0	1	1	1	1	1	1	0	0	1
E_{201}	1	1	1	1	1	0	0	1	1	1	1	1
E_{202}	1	1	1	1	0	0	0	0	1	1	1	1
E_{203}	1	1	1	1	0	0	0	0	1	1	1	1
E_{204}	1	1	1	1	1	0	0	1	1	1	1	1

이 논문에서는 문제를 간단히 하기 위해서 한 명의 운영자(*Operator*)와 3명의 손님(*Customer0, Customer1, Customer2*), 2개의 펌프(*Pump0, Pump1*)로 구성되는 주유소 문제를 고려하며, *Customer0*과 *Customer2*는 *Pump0*을 사용하고 *Customer1*은 *Pump1*을 사용하는 것으로 가정한다. 이 경우에 *Customer0*이 *Pump0*에 대해 요구하는 4개의 이벤트 (E_{001}, E_{002}, E_{003}, E_{004}), *Customer1*이 *Pump1*에 대해 요구하는 4개의 이벤트 (E_{111}, E_{112}, E_{113}, E_{114}), *Customer2*가 *Pump0*에 대해 요구하는 4개의 이벤트(E_{201}, E_{202}, E_{203}, E_{204})로 인하여 12개의 동기화 이벤트들이 정의되며, 이 동기화 이벤트들 사이에는 표 1과 같은 의존성이 존재한다. 표 1에서

"1"은 두 동기화 이벤트 사이에 의존성이 있음을 나타내고, "0"은 두 동기화 이벤트 사이에 의존성이 없음을 나타낸다.

4.2 동치 집합 오토마타 생성

오토마타 생성기는 테스트 시퀀스와 이벤트의 의존성 테이블을 입력으로 받아서 테스트 시퀀스와 동일한 행위를 하는 동기화 시퀀스들만을 받아들이는 동치 집합 오토마타를 생성한다. 이 과정은 (1) 이벤트의 의존성 테이블을 이용하여 테스트 시퀀스에 대한 선후 관계 그래프를 생성하는 과정과 (2) 선후 관계 그래프에서 병행적인 이벤트들을 인터리빙 하여 동치 집합 오토마타를 생성하는 과정으로 구성된다 [15]. **알고리즘 1**은 테스트 시퀀스에서 반드시 지켜져야 하는 이벤트의 부분 순서들을 표현하는 선후 관계 그래프를 생성한다.

알고리즘은 선후 관계 그래프의 초기화로 시작한다 (8-12). 선후 관계 그래프에서 "0"은 두 이벤트 사이에 선후 관계가 없음을 의미하며, 양수는 두 이벤트 사이에 부분 순서가 수행 시에 반드시 만족해야 함을 의미한다. 초기화 이후에는 테스트 시퀀스를 뒤에서부터 읽어서 변수 *pre_event*에 저장하고 (14-18), *pre_event* 이후에 위치하는 이벤트를 변수 *post_evnet*에 저장한다 (19-23). *pre_event*와 *post_event* 사이에 의존성이 있지만 선후 관계 그래프에 표현되지 않은 경우에는 선후 관계 그래프에 의존성을 추가하며 (24-25), 또한 그로 인해 야기되는 모든 의존성 - *pre_event*로부터의 패스 - 을 선후 관계 그래프에 추가한다 (26-28). 이 과정을 테스트 시퀀스를 구성하는 모든 이벤트에 반복적으로 적용함으로써 선후 관계 그래프를 생성한다. 예를 들어, 가스 충전소 예제에서 테스트 시퀀스 [E_{001}, E_{111}, E_{112}, E_{113}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{114}, E_{203}, E_{204}]가 주어지면 **알고리즘 1**은 표 2와 같은 선후 관계 그래프를 생성한다. 그림 5는 표 2의 내용을 그래프 형태로 표현한 것으로, 그래프의 노드와 에지는 각각 동기화 이벤트와 이벤트 사이에서 반드시 만족해야 하는 부분 순서를 표현한다.

표 2 알고리즘 1에 의해서 테스트 시퀀스 [E_{001}, E_{111}, E_{112}, E_{113}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{114}, E_{203}, E_{204}] 로부터 생성된 선후 관계 그래프

Event	E_{001}	E_{002}	E_{003}	E_{004}	E_{111}	E_{112}	E_{113}	E_{114}	E_{201}	E_{202}	E_{203}	E_{204}
E_{001}	0	1	2	3	1	2	3	4	5	5	6	6
E_{002}	0	0	1	2	0	0	1	2	3	3	4	4
E_{003}	0	0	0	1	0	0	0	0	0	2	0	3
E_{004}	0	0	0	0	0	0	0	0	0	1	0	2
E_{111}	0	0	0	0	0	1	2	3	4	4	5	5
E_{112}	0	0	0	0	0	0	1	2	3	3	4	4
E_{113}	0	0	0	0	0	0	0	1	2	2	3	3
E_{114}	0	0	0	0	0	0	0	0	1	1	2	2
E_{201}	0	0	0	0	0	0	0	0	0	0	1	2
E_{202}	0	0	0	0	0	0	0	0	0	0	0	1
E_{203}	0	0	0	0	0	0	0	0	0	0	0	1
E_{204}	0	0	0	0	0	0	0	0	0	0	0	0

알고리즘 1 (선후 관계 그래프 생성)

```
 1: Global Variables
 2:     int eve_count;        // 이벤트의 개수 (=12)
 3:     int seq_count;        // 테스트 시퀀스의 길이
 4:     String Events[eve_count];                    // 동기화 이벤트
 5:     String Sequence[seq_event];                  // 테스트 시퀀스
 6:     int Dependency[eve_count][eve_count];        // 의존성 테이블
 7: private void get_Graph ()   {
 8:     PrecedenceGraph = new int[seq_count][seq_count];
 9:     for( int i=0 ; i<seq_count ; i++)   {
10:         for( int j=0 ; j<seq_count ; j++)   {
11:             PrecedenceGraph[i][j] = 0;
12:     }  }
13:     int pre_event=0, post_event=0;          // 의존 관계를 확인할 두 이벤트의 인덱스
14:     for( int i=seq_count-2 ; i>=0 ; i--)   {
            // 선행 이벤트의 선택
15:         for( int k=0 ; k<eve_count ; k++)   {
16:             if( Sequence[i].equals(Events[k]) )
17:                 pre_event = k;
18:         }
              // 후행 이벤트의 선택
19:         for( int j=i+1 ; j<seq_count ; j++)   {
20:             for( int k=0 ; k<eve_count ; k++)   {
21:                 if( Sequence[j].equals(Events[k]) )
22:                     post_event = k;
23:             }
              // 이벤트 사이에 의존성이 있으면 그래프에 추가
24:             if( Dependency[pre_event][post_event] == 1 && PrecedenceGraph[i][j] == 0 )
25:                 PrecedenceGraph[i][j] = 1;
26:                 for( int l=j+1 ; l<seq_count ; l++ )   {
27:                     if( PrecedenceGraph[j][l] != 0 )
28:                         PrecedenceGraph[i][l] = PrecedenceGraph[j][l]+1;
29: }  }  }  }  }
```

알고리즘 2는 선후 관계 그래프로부터 동치 집합 오토마타를 생성한다. 생성된 동치 집합 오토마타는 테스트 시퀀스와 동일한 행위를 하는 모든 시퀀스들을 받아들인다. **알고리즘 2**에서 상태는 그 때까지 실행된 이벤트의 집합으로 정의하고, 전이는 3-tuple = (원시 상태, 실행 이벤트, 목표 상태) 로 정의한다. 그리고, 오토마타는 전이의 집합으로 구성된다.

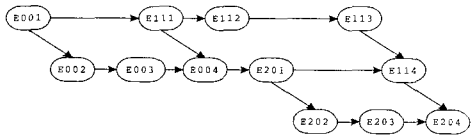

그림 5 표 2를 그래프 형태로 표현한 선후 관계 그래프

알고리즘은 아무 이벤트도 수행되지 않은 초기 상태를 생성하는 것으로 시작하여 (15-18), 현재 상태(*States[current]*)에서 실행될 수 있는 이벤트들 중에서 선행하는 이벤트가 존재하지 않는 이벤트, 즉 가장 먼저 실행될 수 있는 이벤트를 찾는다(22-26). 현재 상태에서 실행 가능한 이벤트를 발견하고, 이 이벤트에 대해서 목표 상태가 존재하지 않는 경우에는 새로운 상태를 생성한다(31-39). 마지막으로 (현재 상태(원시 상태), 실행 가능 이벤트, 목표 상태)의 3-tuple을 새로운 전이로써 오토마타에 추가한다(40-43). 이 과정을 새로이 생성되는 모든 상태에 적용함으로써 테스트 시퀀스의 동치 집합 오토마타를 생성한다. 그림 6은 그림 5의 선후 관계 그래프에 **알고리즘 2**를 적용함으로써 생성되는 동치 집

알고리즘 2 (동치 집합 오토마타 생성)

```
 1: Global Variables
 2:     int st_count, tr_count;              // 상태와 전이의 개수
 3:     int eve_count;        // 이벤트의 개수 (=12)
 4:     int seq_count;                       // 주어진 시퀀스의 길이
 5:     int tr_count:                        // 오토마타에 존재하는 전이의 개수
 6:     String Events[eve_count];  // 동기화 이벤트
 7:     int PrecedenceGraph[eve_count][eve_count];      // 선후 관계 그래프
 8: public void get_Automata()   {
 9:     int sum_col;                                    // 선행 이벤트의 개수
10:     int current, next;                              // 원시 상태와 목표 상태
11:     int[] temp_state = new int[seq_count];
12:     int max_state = power(2, seq_count);
13:     States = new int[max_state][seq_count];          // 오토마타의 상태 = 실행된 이벤트의 집합
14:     Automata = new int[seq_count * max_state][3];  // 동치 집합 오토마타 = 전이의 집합
        // 초기 상태 생성
15:     for( int i = 0 ; i < seq_count ; i++ )
16:         States[0][i] = 0;
17:     current = 0;
18:     st_count = 1;
19:     while( current < st_count )   {
20:         for ( int i = 0 ; i < seq_count ; i++)   {
21:             sum_col = 0;
22:             if (States[current][i] == 0)   {            // i 번째 이벤트가 수행되지 않은 경우
                    // i 번째 이벤트에 대해서 실행되지 않은 이벤트들 중에서 선행 이벤트가 있는지를 검사
23:                 for( int j = 0 ; j < seq_count ; j++ )  {
24:                     if (States[current][j] == 0)
25:                         sum_col = sum_col + PrecedenceGraph[j][i];
26:                 }
                    // I번째 이벤트가 나머지 이벤트들에 대해서 선행 이벤트인 경우 오토마타에 전이 추가
27:                 if ( sum_col == 0 )   {
28:                     copy_array(States[current], temp_state, seq_count);
29:                     temp_state[i] = 1;
30:                     next = -1;
                        // 목표 상태가 이미 존재하는지 검사, 존재하지 않으면 새로운 상태 추가
31:                     for( int k = 0 ; k < st_count ; k++ )   {
32:                         if ( equals_array(temp_state, States[k], seq_count) )
33:                             next = k;
34:                     }
35:                     if (next == -1)   {
36:                         next = st_count;
37:                         copy_array(temp_state, States[st_count], seq_count);
38:                         st_count++;
39:                     }
                        // 전이 추가.
40:                     Automata[tr_count][0] = current;
41:                     Automata[tr_count][1] = i;
42:                     Automata[tr_count][2] = next;
43:                     tr_count++;
44:         } } }
45:         current++;
48: } }
```

합 오토마타이다. 이 오토마타는 테스트 시퀀스[E_{001}, E_{111}, E_{112}, E_{113}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{114}, E_{203}, E_{204}]와 동일한 행위를 하는 모든 시퀀스를 받아들이며, 이 오토마타에 의해서 받아들여지는 모든 시퀀스들은 프로그램에서 테스트 시퀀스와 동일한 행위를 보인다[15].

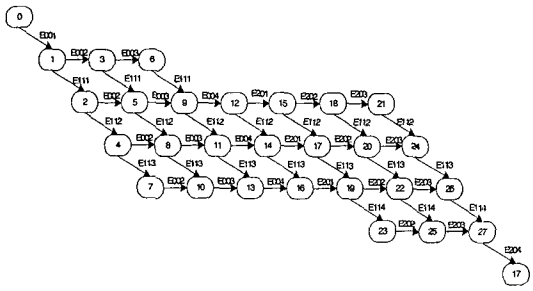

그림 6 그림 5의 선후 관계 그래프로부터 생성된 동치 집합 오토마타

4.3 재연 제어기를 이용한 프로그램 변환

오토마타 기반의 결정적 테스팅에서는 동치 집합 오토마타를 기준으로 프로그램의 수행을 제어하기 위해서, 재연 제어기를 이용하여 프로그램을 변환한다. 변환된 프로그램에서 재연 제어기는 동치 집합 오토마타를 기준으로 동기화 이벤트들의 수행을 허용하거나 보류함으로써 프로그램의 수행 순서를 제어하며, 프로그램 내에서 실제로 실행되는 이벤트의 순서를 기록한다. 그리고, 원래의 Java 다중 스레드 프로그램에서는 스레드의 이벤트 수행 전후에 재연 제어기와의 통신을 위한 코드를 삽입한다. 그림 7은 가스 충전소 예제에 오토마타 기반의 결정적 테스팅 방법을 적용하여 변환한 프로그램의 구조이다.

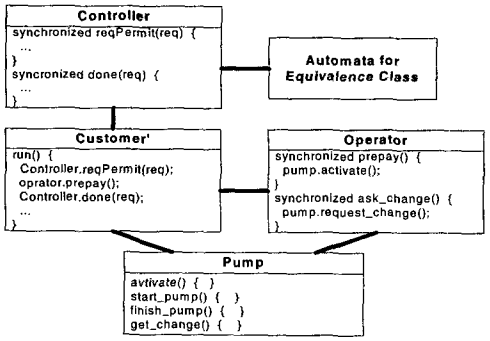

그림 7 오토마타 기반의 결정적 테스팅을 수행하기 위해 변환된 가스 충전소 예제의 구조

Java 다중 스레드 프로그램을 위한 오토마타 기반 테스팅 환경의 재연 제어기는 고전적인 결정적 테스팅의 재연 제어기와 유사한 구조를 가지지만, 동치 집합 오토마타를 기준으로 하여 스레드들의 수행 순서를 제어하는 면에서 차이가 있다. 이를 위하여 재연 제어기는 3개의 주요 메소드를 - *reqPermit(req)*, *done(req)*, *equals(req)* - 포함한다 (그림 8). *reqPermit(req)* 메소드는 스레드에서 요청하는 이벤트 *req*이 동치 집합 오토마타의 현재 상태에서 받아들일 수 있는 입력인 경우에는 스레드의 실행을 허용하지만, 그렇지 않은 경우에는 *wait()* 메소드를 이용해 그 스레드의 실행을 보류한다. *reqPermit(req)* 메소드 내에서 이용하는 *equals(req)* 메소드는 동치 집합 오토마타의 현재 상태에서 이벤트 *req*를 받아들일 수 있는지를 판단하고, 받아들일 수 있는 경우에는 이벤트 *req*에 의한 오토마타의 새로운 현재 상태를 찾는다. 마지막으로 *done(req)* 메소드는 *notifyAll()* 메소드를 이용하여 이벤트 *req*의 실행이 끝났음을 다른 스레드들에게 알리고, 보류되고 있던 모든 스레드들을 깨운다. 따라서, 깨어난 스레드들은 오토마타의 새로운 현재 상태에서 다시 *req Permit(req)* 메소드를 호출하게 된다. 한편, 일반적으로 임의의 프로그램에 대해서 어떤 이벤트 시퀀스가 수행 가능한지를 판단하는 문제는 해결 불가능한 문제이기 때문에, *reqPermit(req)* 메소드에서는 하나의 이벤트가 계속해서 수행이 보류되는 경우 - 변수 *cnt*가 보류된 회수이다 - 주어진 프로그램에는 동치 집합 오토마타에 따르는 시퀀스가 존재하지 않는 것으로 판단한다.

```
public class Controller {
  ...
  private static Controller ctrl = new Controller();
  ...
  private Controller() { ... }
  public synchronized static Controller getController() {
        return ctrl;
  }
  public synchronized void reqPermit(Request req) throws InterruptedException {
        long cnt = 0;
        boolean tired = false;
        int inautomata = 0;
        while (count != next || (tired && (inautomata = equals(req)) == 0)) {
          wait(500);
          if (cnt < 20)
            cnt++;
          else
            tired = true;
        }
        ...
  }
  public synchronized void done(Request req) throws InterruptedException {
        count++;
        notifyAll();
  }
  private int equals(Request req) {
        int result = 0;          // 0:error,  1:in automata, 2:not in automata
        ...
        if ( (result = at.feasible(curState, curEvent)) != 0) {
          curState = at.get_nextState(curState, curEvent);
          next++;
        }
        return result;
  }
  ...
}
```

그림 8 Java 다중 스레드 프로그램을 위한 재연 제어기 (*Controller*) 클래스

프로그램 변경에서는 원래의 Java 다중 스레드 프로그램을 구성하는 스레드들에 재연 제어기의 메소드를 호출하는 코드들을 추가함으로써 병행적인 동기화 이벤트들의 수행 순서를 제어한다. 그림 9에서 "//*"로 표시되는 코드들이 동기화 이벤트의 수행을 제어하기 위해서 *Customer* 스레드들에 추가되는 코드들이다. 이 코드들의 행위를 살펴보면, *Customer*에서 호출하는 각 동기화 메소드를 호출하기 전에 재연 제어기의 *reqPermit(req)* 메소드를 호출함으로써 실행이 허락받고, 동기화 메소드의 실행한 후에는 *done(req)* 메소드를 호출함으로써 실행을 끝났음을 재연 제어기에 알리게 된다. 이와 같은 프로그램을 변경하는 코드 삽입은 규칙적으로 이루어지기 때문에 자동화가 용이하다.

```
public class Customer extends Thread {
  ...
  public void run() {
    try {
        Request req;                                          // added code
        Controller ctrl = Controller.getController();         // added code
        opr = Operator.get_operator();

        while(!opr.is_available(pumpNumber));
        req = new Request(threadID, pump.getID(), PRE_PAY);   // added code
        ctrl.reqPermit(req);                                  // added code
        opr.prepay(myNumber, myMoney, pumpNumber, pump);
        ctrl.done(req);                                       // added code

        req = new Request(threadID, pump.getID(), START_PUMPING);  // added code
        ctrl.reqPermit(req);                                  // added code
        pump.start_pumping();
        ctrl.done(req);                                       // added code

        req = new Request(threadID, pump.getID(), FINISH_PUMPING);  // added code
        ctrl.reqPermit(req);                                  // added code
        pump.finish_pumping();
        ctrl.done(req);                                       // added code

        req = new Request(threadID, pump.getID(), ASK_CHANGE);   // added code
        ctrl.reqPermit(req);                                  // added code
        int change = opr.ask_change(myNumber, pumpNumber, pump);
        ctrl.done(req);                                       // added code
    }
    catch (InterruptedException e) {};
  }
  ...
}
```

그림 9 재연 제어기와의 통신을 위해서 변경된 *Customer* 스레드

5. 오토마타 기반 테스팅의 적용 예

오토마타 기반의 결정적 테스팅의 결과를 효과적으로 기술하기 위해서, 우리는 가스 충전소 예제에 대해서 펌프가 모두 활성화 된 상황에서는 항상 *Pump0*이 *Pump1*보다 먼저 충전을 시작한다고 가정한다. 그림 10은 위의 가정을 원시 코드에 반영한 것으로, 이는 명세에 기술된 내용에 대해서 설계 단계의 결정에 의해서 명세 중의 일부만을 구현한 것으로 볼 수 있다. 그러나, 이와 같은 경우에도 가스 충전소에서는 모든 손님들에게 서비스를 제공할 수 있기 때문에 프로그램 상에 오류는 존재하지 않는 것으로 볼 수 있다. 그러나, 그림 10의 프로그램에서 동기화 시퀀스 [E_{001}, E_{111}, E_{112}, E_{113}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{114}, E_{203}, E_{204}]를 실행하는 경우를 생각해 보면, E_{001} 과 E_{111}에 의해서 *Pump0*과 *Pump1*이 모두 활성화되기 때문에 E_{112}가 먼저 실행될 수 없으며, 따라서 주어진 시퀀스와 동일한 시퀀스는 그림 10의 프로그램에서 실행되지 않는다. 이와 같은 경우에, 고전적인 결정적 테스팅에서는 주어진 테스트 시퀀스는 실행이 불가능한 것으로 판단하며, 다른 테스트 시퀀스를 이용하여 추가적인 테스팅을 시도하도록 하거나, 혹은 프로그램에 오류가 있는 것으로 결론을 내린다. 그러나, 오토마타 기반의 결정적 테스팅에서는 테스트 시퀀스와 동일한 행위를 하는 다른 시퀀스가 존재하는 경우에는 이 시퀀스를 이용하여 프로그램을 수행한다.

```
public class Customer extends Thread  {
  ...
  public void run()  {
    ...
    opName = Controller.PRE_PAY;
    opr.prepay(myNumber, myMoney, PumpNumber, pump);
    ...

    // 설계 단계의 결정에 의해 행위가 축소된 코드
    while ( (pump.getID() == 1) &&
            pump.act_state() && otherPump.act_state() &&
            !pump.pump_state() && !otherPump.pump_state() ) ;

    opName = Controller.START_PUMPING;
    pump.start_pumping();
    ...
  }
  ...
}
```

그림 10 설계 단계의 결정에 의해서 행위가 축소된 *Customer* 스레드

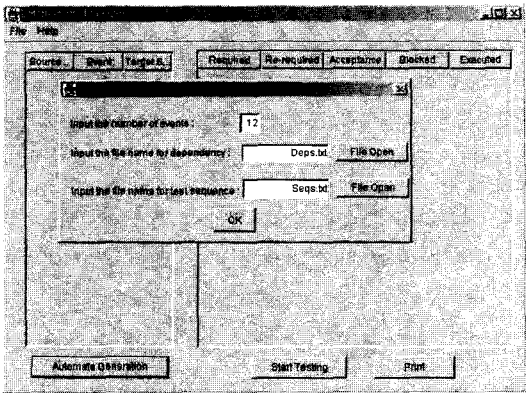

그림 11 오토마타 기반 테스팅 환경에서 테스팅 수행 부분을 위한 지원 도구의 인터페이스

Java 다중 스레드 프로그램에 오토마타 기반의 결정적 테스팅 방법을 적용한 결과를 알기 위해서 테스트

수행 지원 도구인 *Deterministic Executor*를 구현하여 이용한다. *Deterministic Executor*는 입력으로써 동기화 이벤트의 개수와 이벤트들의 의존성 테이블, 테스트 시퀀스를 받지만, 프로그램의 변경 및 재연 제어기와의 연결은 수동으로 이루어진다(그림 11). 도구 내부에서는 먼저 오토마타 생성기를 이용하여 동치 집합 오토마타를 생성하고, 생성된 오토마타와 재연 제어기를 이용하여 변환된 프로그램을 실행함으로써 오토마타 기반의 결정적 테스팅의 결과를 출력한다. 그림 12는 위의 가정으로 행위가 축소된 가스 충전소 예제에 대해서 테스트 시퀀스 [E_{001}, E_{111}, E_{112}, E_{113}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{114}, E_{203}, E_{204}]를 이용하여 오토마타 기반의 결정적 테스팅을 수행한 결과를 출력한다.

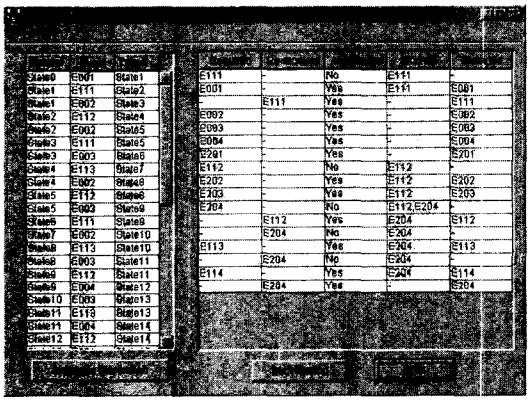

그림 12 가스 충전소 예제에 오토마타 기반의 결정적 테스팅을 적용한 결과

그림 12의 결과를 보면, 왼쪽의 화면에서는 전이들의 집합으로써 오토마타 생성기에 의해서 생성된 동치 집합 오토마타를 표현하고 있다. 이 오토마타는 그림 6의 동치 집합 오토마타와 동일한 오토마타를 표현하고 있다. 오른쪽의 오토마타 기반의 결정적 테스팅의 실행 결과 화면에서, "*Required Event*"는 스레드 *Customer*에서 재연 제어기에 실행을 요청하는 이벤트이며, "*Re-required Event*"는 보류된 이벤트들 중에서 다시 재연 제어기에 실행을 요청하는 이벤트들이다. 그리고, "*Blocked Event*"는 실행이 보류되고 있는 이벤트들이고, "*Executed Event*"는 프로그램에서 현재 실행된 이벤트이다. 따라서, 그림 12에서 8번째 줄에서 E_{112}는 가정에 의해서(E_{111}, E_{201}에 의해서) 실행이 보류되며, E_{202}가 수행된 이후에 수행이 가능해진다. 결론적으로 가스 충전소 프로그램에서는 주어진 테스트 시퀀스에 대해서 [E_{001}, E_{111}, E_{002}, E_{003}, E_{004}, E_{201}, E_{202}, E_{203}, E_{112}, E_{113}, E_{114}, E_{204}]의 순서로 수행된다. 이 수행된 시퀀스는 주어진 테스트 시퀀스와 순서는 다르지만 같은 행위를 가지는 시퀀스이다. 따라서, 행위가 축소된 가스 충전소 프로그램에서는 주어진 테스트 시퀀스에 대한 행위를 지원하고 있다는 결론을 내릴 수 있다.

한편, 주어진 테스트 시퀀스에 대해서 동일한 행위를 하는 다른 시퀀스가 프로그램에서 수행 가능하지 않은 경우에는 그림 13에서와 같이 프로그램의 실행이 중지된다. 이 경우는 주어진 테스트 시퀀스뿐만 아니라 그와 동일한 행위를 하는 어떤 시퀀스도 프로그램 내에서 수행되지 않음을 나타낸다. 따라서, 오토마타 기반의 결정적 테스팅에서는 한 번의 테스트 시퀀스의 수행만으로 Java 다중 스레드 프로그램이 주어진 테스트 시퀀스에 해당하는 행위를 수행할 수 있는지를 판단할 수 있다.

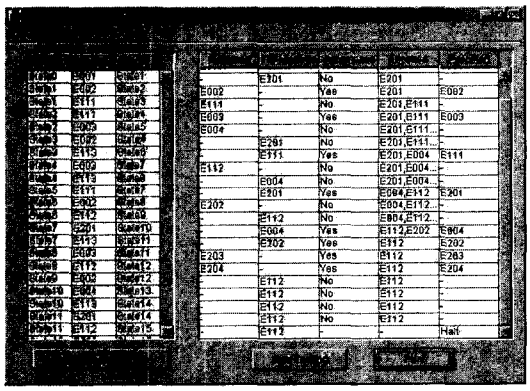

그림 13 테스트 시퀀스와 동일한 행위를 하는 다른 시퀀스가 존재하지 않는 경우의 결과

6. 결론 및 향후 연구 방향

이 논문에서는 병행 프로그램을 테스트하는 새로운 방법인 오토마타 기반의 결정적 테스팅을 위한 테스팅 환경을 제안하고, 오토마타 기반의 결정적 테스팅을 수행하기 위한 지원 도구의 설계 및 구현 방법에 대해서 제안하였다. 테스트 수행 도구에서는 주어진 Java 다중 스레드 프로그램에 오토마타 기반의 결정적 테스팅을 적용한 변환된 프로그램을 생성하고 이를 수행함으로써 테스트의 결과를 출력한다. 이를 위하여 동치 집합 오토마타를 생성하는 *오토마타 생성기*와 Java 다중 스레드 프로그램의 수행을 제어하기 위한 *재연 제어기*를 설계하고 구현하였다.

고전적인 결정적 테스팅 방법에서는 한 번의 수행에서 하나의 테스트 시퀀스에 대한 수행 가능성을 판단하는데 반해서, 오토마타 기반의 결정적 테스팅 방법에서는 테스트 시퀀스의 동치 집합 오토마타를 이용하여 한 번의 수행 동안에 테스트 시퀀스와 동일한 행위를 가지는 모든 시퀀스에 대해서 프로그램의 수행 가능성을 판단한다. 따라서, 오토마타 기반의 결정적 테스팅은 설계 단계에서의 결정에 의해서 동일한 행위에 대해서 명세의 부분적인 구현을 한 병행 프로그램을 테스트하는 방법으로 적합하며, 이 논문의 가스 충전소 실행 예제를 통해서 그 사실을 확인할 수 있다.

향후 연구 방향으로는 재연 제어기와의 통신을 위한 프로그램 변경과 변경된 프로그램을 재연 제어기와 통합할 수 있는 자동화된 도구 구현이 필요하며, 이를 이용하면 완전한 테스트 수행 도구를 개발할 수 있다. 또한 오토마타 기반 테스팅 환경의 앞부분을 구성하는 테스트 케이스 생성 부분을 위한 지원 도구가 필요하다. 테스트 케이스 생성 도구에는 테스트 시퀀스 생성기와 의존성 추출기가 포함된다. 테스트 시퀀스 생성기에서는 병행 프로그램이 가지는 각각의 행위에 대해서, 그 행위를 대표할 수 있는 하나의 테스트 시퀀스만을 생성하게 된다. 마지막으로 이러한 도구들을 이용하여 Java 다중 스레드 프로그램을 위한 자동화되고 통합된 오토마타 기반의 테스팅 환경을 구성할 필요가 있다.

참 고 문 헌

[1] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural Testing of Concurrent Programs", *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 206-215, Mar. 1992

[2] K. C. Tai, "Testing Concurrent Programs", *Proceedings of 9th International Computer Software and Applications Conference (COMSAC'85)*, pp. 310-317, 1985

[3] R. H. Carver and K. C. Tai, "Replay and Testing for Concurrent Programs", *IEEE Software*, pp. 66-74, Mar. 1991

[4] P. A. Emrath, S. Ghosh and D. A. Padua, "Detecting Nondeterminacy in Parallel Programs", *IEEE Software*, pp. 69-77, Jan. 1992

[5] 정인상, "Java 다중 스레드 프로그램의 결정적 수행을 위한 프로그램 변환 방법", *정보과학회 논문지*, 제 27권, 제 6호, pp.607-617, Jun. 1999

[6] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transaction on Computers*, vol. C-36, no. 4, pp. 471-482, 1987

[7] K. C. Tai, R. H. Carver, and E. E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution", *IEEE Transaction on Software Engineering*, vol. 17, no. 1, pp. 45-63, Jan. 1991

[8] H. W. Sohn, D. C. Kung, and P. Hsia, "State-based Reproducible Testing for CORBA Applications", *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pp. 24-35, 1999

[9] X. Cai and J. Chen, "Control of Nondeterminism in Testing Distributed Multithreaded Programs", *Proceedings of 1st Asia-Pacific Conference on Quality Software*, pp. 29-38, 2000

[10] G. V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing", *Proceedings of International Symposium on Software Testing and Analysis*, pp. 109-124, 1994

[11] R. H. Carver and K. C. Tai, "Use of Sequencing Constraints for Specification-based Testing of Concurrent Programs", *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 471-490, Jun. 1998

[12] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee, "Testing of Concurrent Programs Based on Message Sequence Charts", *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 99)*, pp. 72-82, May 1999

[13] I. S. Chung and B. M. Kim, "Yet Another Approach to Deterministic Execution Testing for Distributed Programs", *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pp.55-60, 2000.

[14] I. S. Chung, B. M. Kim, and H. S. Kim, "A New Approach to Deterministic Execution Testing For Concurrent Programs", *International Workshop on Distributed System Validation and Verification*, pp. E-59 E-66, 2000

[15] 정인상, 김병만, 김현수, "오토마타기반의 병행 프로그램을 위한 결정적 수행 테스팅 방법", *정보과학회 논문지*, 제 28권, 제 10호, Oct. 2001

[16] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs", *IEEE Software*, vol. 2, no. 2, pp. 47-57, Mar. 1985

서 희 석

1998년 한국과학기술원 전산학과 학사. 2000년 한국과학기술원 전산학과 석사. 2000년 ~ 현재 한국과학기술원 전자전산학과 전산학 전공 박사 과정 중. 관심분야는 명세 기반 테스팅, 병행 프로그램 테스팅, 아키텍처 기반 테스팅

정 인 상

정보과학회논문지 : 소프트웨어 및 응용 제 29 권 제 8 호 참조

김 병 만

1987년 서울대학교 컴퓨터공학과 학사. 1989년 한국과학기술원 전산학과 석사. 1992년 한국과학기술원 전산학과 박사. 1992년 ~ 현재 금오공과대학교 부교수. 1998년 ~ 1999년 미국 UC, Irvine 대학 방문교수. 관심분야는 프로그램 테스팅 및 검증, 인공지능, 정보검색

권 용 래

정보과학회논문지 : 소프트웨어 및 응용 제 29 권 제 8 호 참조