오토마타기반의 병행 프로그램을 위한 결정적 수행 테스팅 기법

(Deterministic Execution Testing for Concurrent Programs based on Automata)

정 인 상 † 김 병 만 †† 김 현 수 †††
(In Sang Chung) (Byeong Man Kim) (Hyeon Soo Kim)

요 약 이 논문에서는 병행 프로그램을 테스팅하기 위해 오토마타에 기반을 둔 새로운 결정적 수행 방법을 제시한다. 이 논문에서 제안한 방법은 부분 순서 방법에서 상태 폭발 문제를 해결하기 위해 사용되는 이벤트 독립 개념을 사용하여 주어진 이벤트 시퀀스와 의미적으로 동일한 시퀀스들을 받아들일 수 있는 오토마타를 구축한다. 따라서 프로그램이 주어진 시퀀스를 직접적으로 구현하지 않았다 할지라도 그와 의미적으로 동일한 대안 시퀀스가 존재한다면 프로그램의 실행을 허락한다. 이 방법의 장점은 프로그램이 명세에 기술된 대로 정확하게 구현되어 있지 않는 상황에서도 적용할 수 있다는 점이다.

Abstract In this paper, we present a new approach to deterministic execution for testing concurrent programs. The proposed approach makes use of the notion of event independence which has been used in the partial-order method in order to resolve the state-explosion problem and constructs an automation which accepts all the sequences semantically equivalent to a given sequence. Consequently, we can allow a program to be executed according to event sequences other than the (possibly infeasible) given sequence if they have the same effects on the program's behavior. One advantage of this method is that it can be applied to situations where a program is not exactly implemented as described in the specification.

1. 서 론

일반적으로 병행 프로그램은 비결정성을 내포하고 있기 때문에 동일한 입력 자료에 대해서 반복 수행시 서로 다른 결과를 유발할 수 있다[1]. 이와 같은 비결정적 특징 때문에 병렬 프로그램은 순차 수행 프로그램과 달리 테스팅 및 검증 단계에서 다음과 같은 두가지 어려움에 부딪힌다. 첫째, 한 입력 자료에 대해서 주어진 프로그램을 한 번 수행해보는 것만으로는 오류 여부를 판단하기 어렵다. 순차 수행 프로그램은 동일한 입력에 대해서 항상 동일한 출력을 발생하기 때문에 선택된 입력 자료에 대해서 주어진 프로그램을 한번씩만 수행해 보면 오류 여부를 판단할 수 있다. 그러나 비결정적 병렬 프로그램의 경우에는 주어진 입력 자료에 대해서 프로그램을 수행 했을때 옳은 결과를 출력했다 할지라도 다시 수행했을 때는 잘못된 결과를 낼 수 있다. 둘째, 오류가 발생했을 때 그 오류의 원인을 찾기가 쉽지 않다. 병렬 프로그램 수행시 한번 발생한 오류를 반복적으로 다시 관찰하기가 어렵기 때문에 오류의 원인을 찾기가 어려워진다. 비결정성으로 인한 테스팅의 어려움을 극복하기 위해서는 프로그램의 수행을 제어하는 방법이 필요하다. 이러한 목적을 위해 결정적 수행 테스팅(deterministic execution testing)이라 불리는 몇 가지 테스팅 접근 방법들이 개발되었다[2, 3].

결정적 수행 테스팅(또는 결정적 테스팅)에 관한 대부분의 선행 연구들은 명세와 프로그램간의 동치 관계를

· 본 연구는 정보통신연구진흥원에서 지원하는 2000년 대학기초연구지원사업의 후원으로 연구되었음. (과제번호: 2000-036-02)
† 종신회원 : 한성대학교 컴퓨터공학부 교수
insang@hansung.ac.kr
†† 종신회원 : 금오공과대학교 컴퓨터공학부 교수
bmkim@se.kumoh.ac.kr
††† 비 회 원 : 충남대학교 정보통신공학부 교수
hskim@ce.cnu.ac.kr
논문접수 : 2000년 4월 10일
심사완료 : 2001년 8월 14일

가정하였다[4,5]. 동치 관계는 어떤 입력에 대해 프로그램과 명세가 동일한 동기화 시퀀스(synchronization sequences)들을 포함할 것을 요구한다. 따라서, 만일 프로그램이 어떤 입력에 대해 유효하지 않은 시퀀스를 허락하거나 또는 유효한 시퀀스를 허락하지 않는다면 그 프로그램은 동기화 에러를 갖고 있다고 가정한다.

그러나, 설계과정 중의 결정으로 인해 모든 가능한 유효한(valid) 시퀀스들 중 일부만이 프로그램에서 실현되는 상황을 고려하자. 이 경우에 비록 모든 유효한 시퀀스들이 실현 가능하지는 않지만 해당 프로그램은 여전히 명세를 따른다. 그러므로, 프로그램의 행위에 동일한 효과를 주는 다른 실현 가능한(feasible) 시퀀스가 존재하는지 조사해 볼 필요가 있다. 그러나 고전적인 결정적 테스팅 접근 방법들은 그러한 상황에 대해 적절한 수단을 제공하지 않는다. 왜냐하면 그런 방법들은 모든 실현 가능한 시퀀스들은 유효할 뿐만 아니라 모든 유효한 시퀀스들은 실현 가능하다는 것을 보임으로써 명세와 프로그램간의 동치 관계를 보이려고 하기 때문이다.

이 논문에서는 앞서 언급한 그런 문제를 해결하기 위해 새로운 결정적 수행 방법을 제안한다. 이 방법의 기본적인 생각은 이벤트들의 전 순서(totally-ordered) 시퀀스는 부분 순서(partial-order) 의미론을 보존하고 또한 어떤 조건하에서 의미론적으로 동치인 모든 시퀀스들의 집합으로 확장될 수 있다는 것이다. 우리는 동기화 시퀀스에 대한 동치 관계를 정의하기 위한 의미론적 기초로서 이벤트 독립(event independence)이라는 개념을 사용한다. 따라서 동기화 시퀀스들을 동치 클래스들로 묶을 수 있으며, 또한 어떤 주어진 (아마도 실현 불가능한)시퀀스와 다른 동기화 시퀀스들이 동일한 동치 클래스의 멤버들이라면 그러한 동기화 시퀀스들이 수행되는 것을 허락한다.

이 접근 방법은 병행 프로그램을 수행할 때 발생하는 모든 가능한 이벤트 시퀀스들을 조사하지 않고 병행 프로그램의 특성을 검증하기 위해 개발된 부분 순서 방법(partial-order method)에 기반을 두고 있다[6]. 이 논문에서 제안한 방법과 부분 순서 방법의 주요한 차이점은 부분 순서 방법에서는 각 부분 순서 계산의 확장을 모든 가능한 이벤트 시퀀스 보다는 상태 폭발 문제(state explosion problem)를 해결하기 위해 대표적인 시퀀스들로 제한한다는 것이고, 반면 이 논문에서 제안한 방법에서는 주어진 이벤트 시퀀스(또는 인터리빙)를 그 시퀀스와 동일한 효과를 갖는 모든 시퀀스들을 포함하는 동치 클래스를 형성하기 위해 확장하여 병행 프로그램의 테스팅에 적용한다는 점이다.

이 논문의 구성은 다음과 같다. 먼저 2절에서는 고전적인 결정적 수행 테스팅 방법에 관해 간략하게 논의한다. 3절에서는 이벤트 독립의 개념을 도입한 결정적 수행 테스팅을 위한 새로운 모델을 설명한다. 또한 오토마타 기반 접근 방법을 사용하여 결정적 수행 테스팅의 문제를 어떻게 다루는지를 기술한다. 4절에서는 예제를 이용하여 우리의 방법을 설명하고, 5절에서 관련 연구에 대해 간략히 소개한다. 마지막으로 6절에서 결론 및 향후 연구 방향을 제시한다.

2. 고전적인 결정적 수행 테스팅

고전적인 결정적 테스팅 접근 방법들은 일반적으로 (1) 동기화 시퀀스 선택하기, (2) 선택된 시퀀스에 따라 프로그램 강제 수행하기와 같은 두 개의 주요한 단계들로 이루어진다. 동기화 시퀀스는 프로그램 명세 그리고 (또는) 프로그램 구조를 분석함으로써 얻을 수 있다[5,7,8,9,10]. 또한, 주요 정보와 함께 이벤트 발생 순서를 기록함으로써 앞선 수행으로부터 시퀀스들을 수집하는 것도 가능하다. 어떤 기준에 따라 동기화 시퀀스들을 선택한 후에 선택된 시퀀스에 일치하는 수행 경로를 따르도록 프로그램을 강제화 하는 방법이 필요하다.

주어진 동기화 시퀀스를 재연(replay)하기 위한 전형적인 방법은 어떤 이벤트에 대해 동기화 시퀀스에 정의된 대로 그것이 수행될 다음 이벤트가 될 때까지 그 이벤트의 출현을 지연시키기 위한 어떤 수단을 제공하는 것이다[3]. 동기화 시퀀스에 정의된 순서에 따라 이벤트가 발생함을 보장하기 위하여 우리는 소스 프로그램 P를 약간 다른 프로그램 P′으로 변형하고 제어기 프로세스(controller process)를 도입한다. 제어기 프로세스는 주어진 동기화 시퀀스의 재연을 제어하기 위한 프로세스이다. 예를 들어, 원래 프로그램 P에서 동기화 이벤트를 유발하는 각각의 문장은 변형된 버전 P′에서는 동기화 이벤트를 수행하기 위해 제어기 프로세스의 허락을 요청하기 위한 새롭게 삽입된 문장이 뒤따른다.

정확한 설명을 위해서 Java 다중 쓰레드(multi-threaded) 프로그램에 대한 재연 제어기를 설계한다[11]. Java 프로그램에서는 하나 이상의 쓰레드가 활성화되었을 때 병행성이 발생한다. 재연 제어기는 어떤 쓰레드가 수행될 다음 쓰레드일 때 단지 그 쓰레드만 진행할 수 있도록 허락하고 다른 쓰레드들은 제어기로부터 허락을 얻을 때까지 그들의 수행을 지연(block)시키는 모니터(monitor)를 사용하여 구현될 수 있다. 그러한 모니터는 Java 언어에서 그림 1에서와 같이 동기화 메소드(synchronized methods)를 제공하는 클래스로서 간단하게 표현될 수 있다.

```
class ReplayController {
    ...
    public synchronized void requestPermit(ThreadID id)
throws InterruptedException {
                while (!currentThreadID.equals(id)) wait();
    }
    public synchronized void completeMethod(ThreadID id)
    throws InterruptedException {
                currentThreadID=(ThreadID)threadID.
                elementAt(++currentID);
                notifyAll();
    }
}
```

그림 1 재연 제어기 클래스

이들 메소드들은 쓰레드 *ID*[1] 들의 시퀀스를 포함하는 큐(threadID라 명명된)를 사용한다. 쓰레드가 행위를 수행하기 위해 허락을 요청할 때(다른 객체에서 정의된 메소드의 호출과 같이), 제어기의 "requestPermit" 메소드는 만일 다른 쓰레드들이 제어기 모니터에 대한 잠금(lock)을 획득하고 있지 않다면 큐에 명시된 순서에 따라 그 요청이 허락될 수 있는지 여부를 검사한다. 만일 그 쓰레드의 ID가 시퀀스의 현재 내용과 일치하지 않는다면 그 쓰레드는 지연 대기하게 된다. 그렇지 않으면, 그 쓰레드는 행위를 수행할 수 있게된다. 요청된 행위가 완료된 후에는 지연된 쓰레드들을 깨우기 위해 그 쓰레드는 "completeMethod" 메소드를 호출해야만 한다. 물론, 중요한 행위가 발생하였을 때 제어기에게 알리고, 또한 적절한 행동을 수행하기 위해 제어기로부터 허락을 구할 수 있도록 쓰레드에 오퍼레이션들을 삽입해야 한다. 그림 2는 객체 obj에서 정의된 메소드 "method ()"를 호출하는 간단한 쓰레드의 수정된 버전을 보여준다. 여기서, 객체 controller는 제어기의 인스턴스이다.

```
class TransformedThread extends Thread {
      Controller controller = ReplayController.getController();
      //*제어 객체 controller 생성
      ..
      public void run() {
                ...
                controller.requestPermit(getName());
                //*제어 객체에 obj.method() 실행 요청
                obj.method(...);
                controller.completeMethod(getName());
                //*제어객체에 obj.method() 완료 사실 알림
                ...
      }
}
```

그림 2 쓰레드 변환 예

1) 각각의 쓰레드에서 수행되는 행위들은 전 순서화 즉, 순차화 되었다고 간주되기 때문에, 제어기가 수행 시퀀스를 제어하기 위해 단지 쓰레드 ID만을 유지하는 것으로 충분하다.

이런 종류의 재연 전략은 재연될 동기화 시퀀스가 실현 가능할 것이라고 이미 판명된 상황에서 잘 수행된다. 왜냐하면 고전적인 결정적 테스팅 방법은 앞선 수행으로부터 수집된 (잘못된) 수행 트레이스를 재연하는 데에 주로 관련되었기 때문이다. 물론, 우리는 어떤 입력에 대해 주어진 시퀀스가 실현 가능한지 아닌지에 대해 검사하기 위해 고전적인 결정적 수행 방법을 사용할 수 있다. 만일 주어진 시퀀스에 따라 프로그램을 강제로 수행시킨 것이 성공한다면 그 시퀀스는 실현 가능한 시퀀스라고 결론지을 수 있다. 그렇지 않다면, 가장 최근에 발생한 이벤트와 현재 시각간의 시간 간격이 허용되는 간격보다 클 경우에는 그 시퀀스는 실현 불가능하다고 가정할 수 있다[3]. 그러나, 만일 고려 중인 동기화 시퀀스가 유효하지만 실현 불가능하다고 할 때 고전적인 접근 방법들은 "이 시퀀스는 아마도 실현 불가능할 것이다"라고만 얘기할 뿐 다른 어떤 조치도 취하려고 하지 않는다.

3. 결정적 수행 테스팅을 위한 새로운 모델

이 절에서는 병행 프로그램에서 동기화 오류(synchronization errors)라고 불리는 병행성 관련 오류의 타입을 정의하고 고전적 결정적 테스팅에서 찾으려고 노력한 오류와의 차이를 설명한다. 또한, 새롭게 정의된 동기화 오류를 찾기 위한 해결책을 제시한다. 이러한 목적을 위해서, 이벤트들의 시퀀스에서 동치 관계를 정의하기 위해 의미론적 기초로서 사용된 이벤트 독립의 개념을 도입한다.

3.1 이벤트 독립(Event Independence)

두 이벤트 e_i와 e_j가 있을 때, 만일 그들이 결과나 도달할 수 있는 상태에 영향을 주지 않고 임의의 순서로 수행될 수 있다면, 비정형적으로 그 두 이벤트들은 독립적이라고 얘기되어 지며 $(e_i, e_j) \in I$ 와 같이 표현된다. 이벤트들간의 독립성 개념을 사용하여, 병행 시스템의 정형 검증에서 기본적인 장애로 인식되는 상태 공간 폭발을 피하기 위해 부분 순서 방법이라 불리는 다수의 방법들이 개발되었다. 상태 공간 폭발의 한가지 원인은 n 개의 병행 이벤트들에 대해 $n!$ 개의 인터리빙이 생긴다는 것이다. 병행 독립적인 이벤트들의 인터리빙은 동등하며 어떤 특성을 검증하기 위해서는 단지 대표적인 인터리빙만이 사용된다는 사실은 알려져 있다. 부분 순서 방법에서는 이러한 중복성을 이용하고 도달 가능한 모든 상태보다는 단지 그것들의 부분 집합만을 방문한다. 이벤트 독립성에 대한 좀더 정형적인 정의는 [6]

를 참조하기 바란다.

독립적인 이벤트에 대한 개념을 바탕으로, 이벤트들의 시퀀스에 대한 동치 관계를 정의할 수 있다. 병행 시스템의 임의의 두 이벤트 시퀀스 ω_1과 ω_2가 있을 때, 만일 $(e_i, e_j) \in I$에 대해 $\omega_1 = \omega_{11} e_i e_j \omega_{12}$이고 $\omega_2 = \omega_{11} e_j e_i \omega_{12}$라면 $\omega_1 \equiv \omega_2$라고 정의한다. 이벤트 시퀀스에 대해 동치 관계 $\equiv^*$은 $\equiv$의 반사 전이 폐포(reflexive and transitive closure)이다. 직관적으로, 만일 한 이벤트 시퀀스가 다른 이벤트 시퀀스에서 이웃한 독립적인 이벤트들을 반복적으로 교환함으로써 얻어질 수 있다면 두 이벤트 시퀀스는 동등하다. 이벤트 시퀀스 ω에 동등한 모든 이벤트 시퀀스들의 집합은 $[\omega]$로 표현된다. 그러한 동치 클래스는 Mazurkiewicz 트레이스라 불리어 진다[12].

예를 들어, 이벤트 집합 $E = \{e_1, e_2, e_3\}$를 고려하자. 그리고 e_2는 e_1과 e_3에 대해 종속적이고, e_1과 e_3는 독립적이라고 가정하자. 결과적으로 집합 $D = \{(e_1, e_1), (e_2, e_2), (e_3, e_3), (e_1, e_2), (e_2, e_1), (e_2, e_3), (e_3, e_2)\}$를 얻을 수 있다. 그러면 시퀀스 $\omega = e_1 e_3 e_2$로부터 동치 클래스 $[\omega] = \{e_1 e_3 e_2, e_3 e_1 e_2\}$를 얻을 수 있다. $[\omega]$에서 두 번째 시퀀스 $e_3 e_1 e_2$는 첫 번째 시퀀스 $e_1 e_3 e_2$에서 두 이웃한 이벤트 e_1과 e_3를 교환함으로써 얻을 수 있다.

독립 관계 I는 종속 관계 D를 이용하여 다음과 같이 표현될 수 있다: $I = E^2 \setminus D$. 여기서, E는 이벤트들의 유한 집합을 D는 E 위에서의 이진, 반사(reflexive), 대칭(symmetric) 관계를 나타낸다. 종종 (E, D)를 병행 알파벳이라 한다. 따라서, 종속 관계 D가 주어지면, 이벤트 시퀀스 ω를 포함하는 트레이스 $[\omega]_{(E,D)}$는 시퀀스 ω 중의 하나와 병행 알파벳(E, D)에 의해 완전하게 특징지어 진다. 그러나, 우리는 모호함이 없다면 $[\omega]_{(E,D)}$를 $[\omega]$로 나타낸다.

만일 어떤 상태가 동치 클래스 $[\omega]$ 내의 이벤트들의 시퀀스 w_1에 의해 도달 가능하다면, 그 상태는 또 다른 시퀀스 $w_2 \in [\omega]$에 의해서도 또한 도달 가능하다. 따라서 다음과 같이 정리할 수 있다.

정리 1] ([6]) s를 병행 시스템에서 상태라고 하자. 만일 $s \xRightarrow{\omega_1} s_1$이고 $s \xRightarrow{\omega_2} s_2$, 그리고 $[\omega_1] = [\omega_2]$이면, $s_1 = s_2$이다. 여기서 $s \xRightarrow{\omega} s'$는 이벤트들의 유한 시퀀스 ω를 통해 상태 s에서 s'으로 상태 전이가 발생함을 의미한다.

[증명] $[\omega]$의 정의에 의하여 모든 $\omega' \in [\omega]$를 ω를 구성하는 이벤트들 중에서 인접한 독립적인 이벤트들을 반복적으로 교환함으로써 얻을 수 있다. 따라서, 두 개의 인접한 이벤트들의 순서만 다른 임의의 두 시퀀스 ω_1, ω_2에 대해 만약 $s \xRightarrow{\omega_1} s_1$라면 $s \xRightarrow{\omega_2} s_2$가 만족됨을 보이는 것으로 충분하다. 시퀀스 ω, ω'를 각각 $\omega = t_1 \ldots ab \ldots t_n$, $\omega' = t_1 \ldots ba \ldots t_n$라 가정하자. 이 때 다음이 성립됨을 알 수 있다:

$$s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_i} s_i \xrightarrow{a} s_{i+1} \xrightarrow{b} s_{i+2} \cdots \xrightarrow{t_n} s_n \text{이고}$$

$$s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_i} s_i \xrightarrow{b} s'_{i+1} \xrightarrow{a} s'_{i+2} \cdots \xrightarrow{t_n} s'_n.$$

여기에서 이벤트 a와 b가 독립적이므로 $s_{i-2} = s'_{i+2}$가 성립한다. 이벤트 시퀀스 ω_1에서 상태 s_{i+2}로부터의 전이들이 ω_2에서 상태 s'_{i+2}로 부터의 전이들과 동일하기 때문에 $s_n = s'_n$이 만족됨을 알 수 있다. ■

이 정리는 만일 시퀀스 w_1이 유효하지만 실현 가능하지 않다면, $w_2 \in [w_1]$일 때 결정적 수행을 위해 w_1의 복사본으로서 시퀀스 w_2를 사용할 수 있음을 의미한다.

3.2 병행 프로그램에서 동기화 오류

일반적으로, 고전적인 결정적 수행 테스팅은 동기화 동치 관계를 기반으로 한다. 여기서는 명세의 동기화 이벤트 시퀀스와 동일한 시퀀스를 구현이 포함하는 지를 검사하게 된다[1]. 따라서 만일 어떤 입력 x에 대해 "valid(S, x)"와 "feasible(P, x)가 같지 않다면 프로그램은 동기화 오류를 갖고 있다고 간주된다. 여기서 "valid(S, x)"는 입력 x에 대한 명세 S의 동기화 시퀀스들의 집합을 나타낸다. "feasible(P, x)"는 입력 x에 대한 프로그램 P의 실현 가능한 동기화 시퀀스들의 집합을 나타낸다.

그러나 실제로, 동기화 동치 관계를 기반으로 한 결정적 수행 테스팅을 적용하기 어려운 상황들이 많이 존재한다. 병행 프로그램에 대한 대부분의 명세 언어들은 동기화 이벤트들간의 순서 제약 조건을 기술하기 위한 부분 순서 의미론을 갖는다. 동치 관계를 기반으로 한 이전의 테스팅 프레임워크에서는 부분 순서 집합으로부터 유도된 모든 전 순서 시퀀스들이 프로그램 수행 중에 관찰되어야 한다. 이러한 접근 방법에서 한가지 주요한 문제는 설계 결정에 따라 단지 전 순서 시퀀스들의 부분 집합만이 구현될 수 있다는 것이다[7].

예를 들어, 전화 통화 예제의 호 시도 프로세스를 고려하자. 실제로는 피호출자가 링(ringing) 톤을 듣기에 앞서서 호출자가 프리(free) 톤을 듣던지 혹은 그 반대이던지 문제가 되지 않는다. 이런 종류의 비결정성은 두 시나리오 중 하나만을 실현함으로써 대응되는 프로그램

에서 순차화될 수 있다. 왜냐하면 두 병행 이벤트들의 순서는 중요하지 않기 때문이다. 그러나 그러한 상황에 대해 고전적인 결정적 수행 테스팅 접근 방법들은 만일 프로그램이 기대하는 모든 시나리오들을 보여주지 못한다면 그 프로그램은 동기화 오류를 갖는다고 가정한다[4,13,14,15].

우리는 위에서 토론한 내용들을 바탕으로 동기화 동치 관계는 실제 상황에서는 효과적이지 못하다는 결론을 내릴 수 있다. 따라서, 동기화 오류를 좀더 유용하게 사용하기 위해서는 동기화 오류의 새로운 개념을 정의할 필요가 있다.

정의 1] $[\omega]_{(E,D)}$를 명세 S의 입력 x에 대한 동기화 시퀀스 ω와 동일한 효과를 갖는 유효한 동기화 시퀀스들의 집합이라 하자. 그러면, 입력 x에 대해 $[\omega]_{(E,D)} \cap feasible(P,x) \neq \emptyset$ 일 때 프로그램 P는 동기화 오류를 갖는다고 말한다.

따라서 이벤트 E에 대한 종속 관계 D와 입력 x에 대한 프로그램 P의 계산을 명시한 유효한 동기화 이벤트 시퀀스 ω가 주어진다면, 문제는 $[\omega]_{(E,D)}$에 속하는 원소 중 적어도 하나가 입력 x에 대한 프로그램 P의 결정적 수행을 통해 수행될 수 있는지를 검사하는 것이다. 즉, 우리는 다음을 검사하여야 한다: $\exists \omega_i \in [\omega]_{(E,D)}$ such that $\omega_i \in feasible(P,x)$.

트레이스 $[\omega]$의 개념과 순차 프로그램 테스팅을 위해 개발된 revealing subdomain[16] 기법과의 사이에는 유사성이 존재한다. 오류에 대한 reavealing subdomain은 입력 영역(input domain)의 부분 집합으로서 여기서 한 원소의 정확한 수행은 그 부영역 내에 명시된 오류가 존재하지 않음을 보장한다. 즉, 부영역 S에서의 하나의 입력이 정확하게 수행되면 S의 다른 모든 입력에 대해서도 정확하게 수행됨을 보장한다. 이 개념의 의도는 동치 클래스의 모든 원소는 명시된 오류에 대해 동일한 방법으로 취급되어야 한다는 개념으로 프로그램의 입력 영역을 동치 클래스로 분할하려는 것이다. 우리는 정리 1로부터 하나의 트레이스에 속하는 모든 동기화 시퀀스는 프로그램의 행위에 대해 동일한 효과를 가지므로 그들은 동등하다는 것을 알 수 있다. 더욱이 정의 1에서 기술되었듯이, 트레이스는 주어진 시퀀스에 대하여 동기화 오류가 존재하지 않음을 보이는 데에도 사용할 수 있다. 따라서, 우리의 테스팅 접근 방법에서는 동기화 오류가 존재하지 않음을 보장하기 위해서 주어진 시퀀스 ω에 대한 트레이스 $[\omega]$를 구축하는 것은 매우 중요하다. 이 문제는 다음절에서 오토마타 기반 접근 방법을 이용하여 다루어 질 것이다.

3.3 결정적 수행을 위한 오토마타

이 절에서는 정의 1에서 정의된 동기화 오류를 파악하기 위한 방법에 대해 논의한다. 이벤트 E에 대한 동기화 이벤트 시퀀스 ω와 종속 관계 D가 주어지면, 이 방법은 동치 클래스 $[\omega]_{(E,D)}$를 구축하기 위해 오토마타를 기반으로 한 접근 방법을 사용한다.

동기화 이벤트 시퀀스는 유한하고 그것의 각각의 이벤트는 구별된다고 가정한다. 즉, 같은 이름의 두 이벤트들은 다른 것으로 간주된다. 이벤트 시퀀스 $\omega = e_1 e_2 \cdots e_n$이 주어질 때, 만일 $j = i+1$ 이면 두 이벤트 e_i와 e_j는 $e_i \gg_\omega e_j$로 표현되고 만일 $j > i$이면 $e_i >_\omega e_j$로 표현된다.

정의 2] 만일 다음의 조건들을 만족한다면, 직접 선후 관계 $P_{[\omega,D]}$는 E에 속하는 두 이벤트간의 이진 관계로 $(e_i, e_j) \in P_{[\omega,D]}$ 와 같이 정의된다.

1. $\theta \in [\omega]_{(E,D)}$ 에 대해 $e_i \gg_\theta e_j$
2. $e_j \gg_\lambda e_i$를 만족하는 동기화 이벤트 시퀀스 λ가 존재하지 않는다.

비정형적으로, $(e_i, e_j) \in P_{[\omega,D]}$는 ω와 동치 관계인 어떤 동기화 이벤트 시퀀스에 대해 이벤트 e_i후에 이벤트 e_j가 즉각적으로 발생할 수 있고, 또한 이벤트 e_i는 이벤트 e_j후에 발생하지 않음을 의미한다.

정의 3] $P^*_{[\omega,D]}$는 $P_{[\omega,D]}$의 반사 전이 폐포(reflexive and transitive closure)에 의해 정의되며, $P_{[\omega,D]}$의 선후 관계로 참조된다. 즉, $P_{[\omega,D]}$를 포함하는 $P^*_{[\omega,D]}$는 E 상에서 (이벤트 순서에 대해) 가장 작은 반사 전이 관계(reflexive and transitive relation)이다.

따라서, $(e_1, e_2) \in P^*_{[\omega,D]}$는 $[\omega]_{(E,D)}$에 속하는 모든 시퀀스에서 e_1이 e_2에 앞서서 반드시 발생해야 함을 의미한다. 즉, $\theta \in [\omega]_{(E,D)}$에 대해 $e_1 >_\theta e_2$이고, $e_j >_\lambda e_i$을 만족하는 동기화 이벤트 시퀀스 λ가 존재하지 않아야 한다. 따라서, 우리는 선후 관계에 의해 관련되지 않은 이벤트들을 서로 교환함으로써 $[\omega]_{(E,D)}$에 속하는 모든 시퀀스들을 생성할 수 있다.

직접 선후 관계 $P_{[\omega,D]}$가 주어지면, 그 관계를 선후 관계 그래프라 불리는 방향성 그래프 $G_{[\omega,D]} = (N, A)$로 다음과 같이 표현할 수 있다:

- $N = \{n_i\}$, 여기서 n_i는 ω에 속하는 이벤트 e_i에 대응하는 노드이다.

- $A = \{(n_i, n_j) | (event(n_i), event(n_j)) \in P_{[\omega, D]}\}$, 여기서 $event(n)$은 노드 n에 대응되는 이벤트이다. 두 노드 s, t ($\in N$)에 대해, $s \Rightarrow t$는 s에서 t로의 경로가 존재함을 의미하고 $s \not\Rightarrow t$는 그들 사이에 경로가 없음을 의미한다. 또한, 노드 t ($\in N$)에 대한 직접 선행 이벤트와 직접 후행 이벤트를 나타내기 위하여 pre(t)와 suc(t) 표기법을 사용한다. 즉, pre(t)= $\{event(s) | (s, t) \in A\}$이고 suc($t$)= $\{event(s) | (t, s) \in A\}$이다.

이벤트 시퀀스 $\omega = e_1 e_2 \cdots e_n$과 종속 관계 D가 주어지면, 아래의 프로시져(PGPG: Procedure for Generating Precedence Graph)를 통해 직접 선후 관계 $P_{[\omega, D]}$에 대응되는 종속 관계 그래프 $G_{[\omega, D]} = (N, A)$를 만들 수 있다.

1) 각각의 이벤트 e_i ($\in E$)에 대해 하나의 노드 n_i를 생성한다. 이 노드는 $node(e_i)$로 표현한다.
2) $i = n-1 \ldots 1$, $j = i+1 \ldots n$에 대해, 만일 $e_i >_\omega e_j$, $(e_i, e_j) \in D$ 이고 $node(e_i) \not\Rightarrow node(e_j)$이면 방향성 에지 $(node(e_i), node(e_j))$를 첨가한다.

예를 들어, 이벤트 시퀀스 ω = *abcdafgef*와 표 1의 종속 관계 D가 주어지면, 위의 PGPG 프로시져를 적용하여 그림 3의 선후 관계 그래프를 얻을 수 있다.

표 1 종속 관계

Event	a	b	c	d	e	f	g
a	1	0	0	1	0	1	1
b	0	0	0	1	1	1	1
c	0	0	1	1	1	1	1
d	1	1	1	0	1	1	1
e	0	1	1	1	1	0	0
f	1	1	1	1	0	0	1
g	1	1	1	1	0	1	1

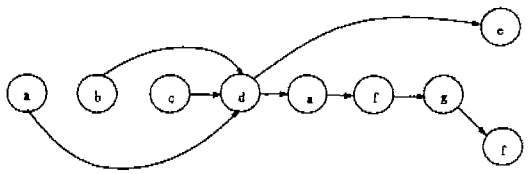

그림 3 선후 관계 그래프

소정리 1] $G_{[\omega, D]} = (N, A)$에 대해, $(node(e_i), node(e_j)) \in A$ *if and only if* $(e_i, e_j) \in P_{[\omega, D]}$.

[증명] *(if 부분)* $P_{[\omega, D]}$의 정의에 의해, a) $e_j \gg_\lambda e_i$ 관계를 만족하는 동기화 시퀀스 λ가 존재하지 않는다. b) $\theta \in [\omega]_{(E, D)}$에 대해 $e_i \gg_\theta e_j$ 관계를 만족한다. 여기서, 문장 a)는 $e_i >_\omega e_j$이고 $(e_i, e_j) \in D$ 관계를 만족함을 의미한다. 또한, b)는 $e_i >_\omega e' >_\omega e_j$와 $(e_i, e') \in D$, $(e', e_j) \in D$ 관계를 만족하는 이벤트 e'이 존재하지 않음을 의미한다. PGPG 프로시져는 만일 $e_i >_\omega e_j$, $(e_i, e_j) \in D$ 이고 $node(e_i) \not\Rightarrow node(e_j)$이면, 두 노드 $node(e_i)$와 $node(e_j)$간의 방향성 에지를 생성한다. 따라서, $(e_i, e_j) \in P_{[\omega, D]}$이면 $(node(e_i), node(e_j)) \in A$ 이다.

(only if 부분) PGPG 프로시져는 만일 $e_i >_\omega e_j$, $(e_i, e_j) \in D$ 이고 $node(e_i) \not\Rightarrow node(e_j)$이면, 두 노드 $node(e_i)$와 $node(e_j)$간의 방향성 에지를 생성한다. $e_i >_\omega e_j$와 $(e_i, e_j) \in D$ 관계에 의해, e_i는 항상 e_j에 앞서 발생한다. 또한, $node(e_i) \not\Rightarrow node(e_j)$ 관계에 의해, e_j는 e_i 후에 즉각적으로 발생할 수 있다. 따라서 $(node(e_i), node(e_j)) \in A$이면 $(e_i, e_j) \in P_{[\omega, D]}$이다. ■

$[\omega]_{(E, D)}$에 속하는 동기화 시퀀스를 받아들이는 오토마타는 (Q, I, F, T)로 표현된다. 여기서 Q는 상태들의 집합을 I ($\subset Q$)와 F ($\subset Q$)는 각각 초기 상태들의 집합과 최종 상태들의 집합을 나타낸다. T ($= Q \times E \times Q$)는 전이들의 집합이다. 상태 t ($\in Q$)는 그 상태에서 발생할 수 있는 이벤트들의 집합으로 표현된다. 선후 관계 그래프 $G_{[\omega, D]} = (N, A)$가 주어졌을 때, 초기 상태 집합 I는 $\{event(n) | \text{pre}(n) = \{\ \}$ 모든 $n \in N$에 대해 $\}$로 정의된다. 상태간의 전이는 현재 상태에서의 이벤트들 중의 하나를 선택함으로써 수행된다. 즉, 현재 상태에서 선택된 이벤트를 제거하고 그 선택된 이벤트와 선후 관계에 의해 관련된 이벤트들을 첨가함으로써 다음 상태로 진행할 수 있다. 그러나 선후 관계로 관련된 두 이벤트가 어떤 상태에서 동시에 존재할 수 있다. 이 경우에는 이벤트 쌍 $(e_i, e_j) \in P^*_{[\omega, D]}$의 경우에 이벤트 e_j가 이벤트 e_i에 앞서 발생할 수 있기 때문에 이벤트 순서 제약조건이 위배된다. 이 문제를 강조하기 위하여 우리는 부가 함수를 도입한다.

정의 4] 선후 관계 그래프 $G_{[\omega, D]} = (N, A)$ 가 있을 때, s를 $G_{[\omega, D]}$에 대응되는 오토마타의한 상태라 하고 이벤트 집합 $E(s) = \{e_1, e_2, \ldots, e_n\}$로 표현된다고 하자. 만일 e가 상태 s에서 선택된 이벤트라면, conflict(s, e)는 다음과 같이 정의된다: conflict(s, e)= $\{ event(n) | n \in$ suc($node(e)$), $\exists t \in E(s) - \{e\}$ such that $node(t) \Rightarrow n$ $\}$.

"conflict(*s*, *e*)"는 이벤트 *e*에 의해 진행된 *s*의 다음 상태에서 배제되어야 하는 이벤트들의 집합이다. 오토마타의 각 상태는 그 상태에서의 이벤트들 중의 하나를 선택함으로써 다음 상태로 진행한다. 따라서, 초기 상태에서 시작하여 전이 규칙을 반복 적용하면 오토마타의 모든 상태와 전이 *Q*, *T*를 구할 수 있다. 최종 상태 집합 *F*는 진출 전이가 없는 즉, 발생할 이벤트가 없는 상태들로 구성된다.

[전이 규칙] 오토마타 *A=(Q,I,F,T)*의 상태 *s*에서 이벤트 *e*가 선택되고 *s*에서 발생할 수 있는 이벤트들의 집합을 *V*라 가정하자. 그러면 다음 상태 $t(\in Q)$는 $W=V-\{e\}+\mathrm{suc}(node(e))-\mathrm{conflict}(s, e)$로 표현되고, 전이 (s, t)에 대한 레이블은 *e*이다.

예를 들어, 그림 3의 선후 관계 그래프를 고려하자. 초기 상태 $s_1=\{a, b, c\}$에서 출발하여 상태 s_2는 다음과 같이 얻을 수 있다: $s_1 \xrightarrow{a} s_2 = \{b, c\}$. 상태 s_2에서 이벤트 *b*와 *c*는 이벤트 *d*에 앞서서 반드시 발생해야 하므로 이벤트 *d*는 첨가될 수 없다. 즉, $d \in \mathrm{conflict}(s_1, a)$. 같은 이유로, $s_1 \xrightarrow{b} s_3 = \{a, c\}$, $s_1 \xrightarrow{c} s_4 = \{a, b\}$를 얻을 수 있다. 이와 같이 전이 규칙을 반복 적용하면 그림 4와 같은 오토마타를 얻을 수 있다.

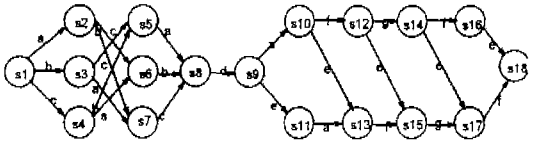

그림 4 오토마타

정리 2] *M*을 $G_{[\omega, D]}$에 대응되는 오토마타라 하자. 또한 *L(M)*은 *M*이 받아들이는 단어들의 집합이라 하자. 그러면 $L(M)=[\omega]_{(E,D)}$.

[증명] $[\omega]_{(E,D)}$는 서로 선후 관계를 갖지 않는 두 이벤트들을 교환함으로써 얻을 수 있는 이벤트 시퀀스들의 집합이다. 만일 그래프 *G*가 선후 관계를 정확하게 표현하고 오토마타 *A*가 *G*에서의 선후 관계 순서를 보전하는 모든 시퀀스들을 받아들인다면 $L(A)=[\omega]_{(E,D)}$이다. 소정리 1에 의하면 선후 관계를 갖는 두 노드 사이에는 경로가 존재한다. 즉, 그래프 *G*는 선후 관계를 정확하게 표현한다. 또한, 오토마타 *A*에서의 상태는 그것의 다음 상태들 중의 하나에서 발생할 수 있는, 물론 그들 사이에 선후 관계가 없는 모든 이벤트들을 포함한다. 그리고 전이 규칙은 각각의 이벤트에 대한 후속자를 생성한다. 따라서 선후 관계를 보전하는 모든 시퀀스들은 오토마타 A에 의해 받아들여진다. ■

4. 예 제

4.1 좌석 예약

하나의 좌석을 예약하기 위하여 그림 6의 클래스 Seat로부터 생성된 공유 객체 'Seat'를 접근하기 위한 두 개의 병행 쓰레드 P_1과 P_2가 있다고 가정하자. P_1과 P_2는 그림 5의 클래스 SeatBooking으로부터 생성된다. 그림 5의 각각의 문장들에는 그것들과 관련된 이벤트를 구별하기 위하여 꼬리표를 붙였다. 쓰레드 P_1로부터 발생한 이벤트들과 P_2로부터 발생한 이벤트들을 구별하기 위하여 각 꼬리표에는 쓰레드 이름이 앞에 붙여진다. 예를 들어, P_1:l_2는 "좌석이 이용 가능한지를 알아보기 위하여 쓰레드 P_1이 객체 Seat에 메시지 read()를 보낸다"는 이벤트를 나타낸다.

```
class SeatBooking implements Runnable {
        boolean status;
                ⋮
        public void run() {
        l1 : status = false;
        l2 : if (seat.read()== false) {
        l3 :        seat.write(true);
        l4 :        status = true;
            }
        }
        ⋮
}
```

그림 5 좌석 예약을 위한 쓰레드 정의

```
class Seat {
        boolean Occupied=false;
        public synchronized boolean read( ) {
        return Occupied;
        }
        public synchronized write(boolean b) {
        Occupied = b;
        }
        ⋮
}
```

그림 6 객체 "seat"에 대한 클래스 정의

좌석 예약 시스템이 유용하기 위해서는 쓰레드 P_1(또는 P_2)이 다른 쓰레드 P_2(또는 P_1)에 의해 좌석이 점유되지 않았을 때는 언제라도 좌석을 예약하는 것이 허락

되어야만 한다. 이것은 다음의 이벤트 시퀀스들을 통해 검사될 수 있다.

- σ_1 = (P_1 : l_1) → (P_1 : l_2) → (P_1 : l_3) → (P_1 : l_4) → (P_2 : l_1) → (P_2 : l_2) (Result:$status_1$ = true, *Occupied* = true, $status_2$ = false).
- σ_2 = (P_2 : l_1) → (P_2 : l_2) → (P_2 : l_3) → (P_2 : l_4) → (P_1 : l_1) → (P_1 : l_2) (Result:$status_1$ = false, *Occupied* = true, $status_2$ = true).

여기서 '$status_i$'는 쓰레드 P_i가 실제로 참조하는 *status* 변수를 나타낸다.

위의 두 시퀀스들은 다른 결과들을 이끌어 낸다: 첫 번째 시퀀스 σ_1은 쓰레드 P_1이 좌석을 예약하고 쓰레드 P_2는 좌석을 예약하지 못한 상황을 보여준다. 그러나 σ_2에서는 쓰레드 P_1가 좌석을 예약하지 못하고 쓰레드 P_2이 좌석을 예약하는 상황을 나타낸다. 이와 같이 프로그램내에 구현되도록 의도된 (비결정적)행위들을 *필수적 비결정성*(obligatory nondeterminacy)이라 불린다[7]. 이와 같이 프로그램에서 구현되어야 할 기본 행위들을 유도한 후에는 이 행위들이 프로그램에서 실제 실현되는지를 검사할 필요가 있다. 이것을 시행하기 위한 직접적인 방법은 기존의 결정적 테스팅 방법을 사용하는 것이다. 좌석 예약 시스템에서 프로그램이 시퀀스에 따라 수행될 수 있도록 강제하기 위하여 우리는 시퀀스 σ_1과 σ_2를 가지고 결정적 테스팅을 사용할 수 있다. 그러나 이 방법은 비록 프로그램이 정확하게 그 시퀀스를 실현하지 못하고 어떤 이유에선가 동일한 결과를 가져오는 다른 시퀀스를 실행한다할지라도 프로그램에 오류가 존재한다고 간주한다.

이 문제를 해결하기 위해 이 논문에서 제안한 방법은 각 테스트 시퀀스 ω에 대해 시퀀스 ω와 동일한 효과를 갖는 시퀀스들을 받아들이기 위한 오토마타를 구축한다. 오토마타를 구축하기 위해서는 3.3절에서 언급한 것처럼, 만일 이벤트 시퀀스와 그 이벤트들간의 종속 관계를 파악해야 한다. 좌석 예약 시스템의 경우 다음 사항들을 고려하여 종속 관계를 파악한다:

● 동일한 객체에 대한 두 개의 "write" 연산들은 종속적이다. 왜냐하면 그들은 그들의 수행 순서에 종속적인 값을 갖는 객체를 생성하게 되기 때문이다.

● "read"와 "write" 연산들은 종속적이다. 왜냐하면 "read"의 결과는 이 연산들의 순서에 따라 달라질 수 있기 때문이다. 그러나 두 개의 "read" 연산들은 그들의 수행 순서에 독립적으로 동일한 결과를 생성하기 때문에 독립적이다.

표 2 좌석 예약 시스템에 대한 종속 관계

Event	P_1:l_1	P_1:l_2	P_1:l_3	P_1 l_4
P_2:l_1	0	0	0	0
P_2 l_2	0	0	1	0
P_2:l_3	0	1	1	0
P_2:l_4	0	0	0	0

물론, 순차적으로 수행될 것으로 가정된 이벤트들은 종속적이다. 표 2는 좌석 예약 시스템에 대한 이벤트들 간의 종속 관계를 보여준다. 이 종속 관계를 이용하여 3.3 절의 PGPG 프로시져를 적용시키면 그림 7과 같은 종속 그래프를 얻을 수 있다.

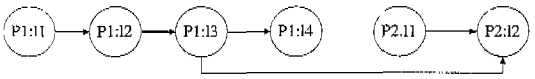

그림 7 이벤트 시퀀스 σ_1을 구성하는 이벤트들에 대한 종속 그래프

종속 관계가 파악되면 앞 절에서 제시한 방법에 따라 오토마타를 구축할 수 있다. 초기 상태 S1={P1:l1, P2:l1}는 종속 그래프에서 진입 에지(incoming edge)가 없는 이벤트들로 구성된다. 상태 S1에서는 두 종류의 이벤트 P1:l1과 P2:l1 모두 발생 가능한데, 만약 P1:l1이 발생하면 다음 상태는 S2 = {P1:l2, P2:l1}이 된다. 상태 S1에서 P2:l1이 발생하면 다음 상태는 S3 = {P1:l1, P2:l2}가 되어야 하나 P2:l2가 conflict(S1, P2:l1)의 원소이기 때문에 이를 제거하여 S3 = {P1:l1}가 된다. P2:l2는 종속 그래프에서 보듯이 P1:l1과 종속 관계에 있다. 따라서, P2:l2는 P1:l1보다 먼저 발생할 수 없고 반드시 P1:l1이 발생 뒤에 발생해야 한다. 상태 S3에서 P2:l2가 제거된 것은 바로 이러한 이유 때문이다. 이러한 과정을 반복 적용하면 시퀀스 σ_1에 대한 오토마타를 구축할 수 있다. 그림 8은 시퀀스 σ_1에 대한 오토마타를 보여준다. 비슷하게, 시퀀스 σ_2에 대한 오토마타도 구축할 수 있다.

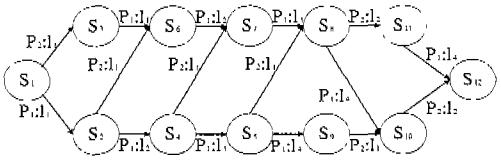

그림 8 이벤트 시퀀스 σ_1에 대한 오토마타

기본적으로 오토마타는 시험 대상 프로그램과 병행적으로 수행되는 제어기이다. 주요한 목적은 오토마타에 의해 받아들여 질 수 있는 임의의 시퀀스를 따르도록 프로그램의 수행을 제어하는 것이다. 이를 위해 2장에서 언급하였듯이 시험 대상 프로그램내의 이벤트를 발생시키는 각 프로그램 문장 앞에 이벤트의 수행을 요청하는 코드(probe)를 삽입하고 문장 바로 뒤에는 문장의 실행이 완료되었음을 제어기에 알려주는 코드를 삽입하여 제어기와 통신할 수 있도록 변환한다. 제어기의 주요 작업은 쓰레드의 사건 시퀀스가 오토마타를 만족하도록 제어하는 것이다. 예를 들어 그림 8의 상태 S2(이미 P1:11 이벤트가 발생하였음을 나타내는 상태)에서는 제어기는 P1:12나 P2:11 이벤트가 발생하기를 기다린다. 만약 P1:12, P2:11 이외의 이벤트가 발생한다면 이 사건을 유발한 쓰레드의 수행을 보류시킨다. 어느 시간동안 P1:12나 P2:11 이벤트가 발생하지 않는다면 σ_1이 대상 프로그램에서 수행이 불가능한 것으로 판단한다. 이는 임의의 프로그램에 대해 주어진 이벤트 시퀀스가 수행 가능한가를 판단하는 문제는 풀 수 없는 문제(undecidable problem)이기 때문이다.

상태 S2에서 쓰레드 P1보다 먼저 쓰레드 P2가 실행되어 이벤트 P2:11를 발생하였다면 제어기는 상태 S6로 전이하여 P1:12가 발생하기를 기다린다. 이 점은 기존의 결정적 테스팅 방법과 차이가 있다. 기존의 방법에서는 주어진 시퀀스(이 경우에는 σ_1)에 따라 프로그램의 수행이 불가능하다면 프로그램에 오류가 있다고 판단한다. 즉, 이벤트 시퀀스 σ_1에서는 P1:11이 발생한 후에 P1:12가 발생하기를 기대하지만 P2:11이 발생하여도 P1:12가 발생하기를 더이상 기다리지 않고 프로그램의 수행을 허락한다. 따라서, 이 논문에서 제안한 방법은 만일 프로그램이 주어진 시퀀스와 다른 시퀀스에 따라 수행된다 할지라도 수행 시퀀스가 오토마타의 한 정로를 이룬다면 프로그램이 기대했던 행위를 수행한다고 판단한다. 이 방법을 적용해 보면, 좌석 예약 시스템은 쓰레드 P_1과 P_2가 좌석을 예약할 수 있는 시퀀스들 즉, σ_1를 포함하는 여러 개의 비결정적 행위들을 구현하였다는 것을 쉽게 알 수 있다.

4.2 전화 통화 시스템

이 예에서는 우리의 접근 방법을 전화 통화 예제 시스템에 적용하여 우리 방법이 어떻게 병행 프로그램 테스팅을 지원하는지 설명한다. MSCs(Message Sequence Charts)[17,18]로 기술된 시스템 명세가 그림 9에 나타나 있다. 그림에는 caller, line_a, alloc_a, region, trunk, alloc_b, line_b의 일곱 개의 프로세스 인스턴스가 있는데 각각은 수직선으로 나타나있다. 프로세스 인스턴스를 가로지르는 육각형은 수행 중에 프로세스가 만족해야 하는 조건을 표현한다.

예를 들어, 프로세스 인스턴스 line_a는 전화 통화 시나리오의 시작 시점에서는 반드시 유휴(idle) 상태여야 한다. 물론, conversation과 end_conversation과 같이 두 개 이상의 프로세스를 가로지르는 육각형은 공유 조건을 의미한다. 두 프로세스를 연결하는 수평 화살표들은 메시지 전달에 의한 통신을 의미한다. 그림 9는 전화 통화 시나리오를 보여주고 있다. 마지막으로, trunk 프로세스에는 수직 점선으로 표현된 부분이 나타나는데 이를 병행영역(coregion)이라 부른다. 한 프로세스 내에 포함된 이벤트들은 수직선상의 위치에 따라 순서를 가지지만 병행영역에 포함된 이벤트들은 순서를 가지지 않는다. 예를 들어, trunk 프로세스는 순서 관계에 의해서 caller 프로세스에게 메시지 tone_off를 보내기 전까지는 메시지 tone_off를 line_b 프로세스에게 보낼 수 없지만, 메시지 free_tone과 메시지 ringing은 병행영역에 포함되어 있으므로 임의의 순서로 보낼 수 있다.

MSCs의 각 이벤트들은 프로세스 이름, 관련된 행위의 타입, 그리고 메시지의 이름을 통하여 구별되어 진다. 예를 들어, "프로세스 t1이 메시지 m1을 전송한다"는 이벤트와 "프로세스 t2가 메시지 m2를 수신한다"는 이벤트는 각각 "t1-send(m1)"과 "t2-recv(m2)"로 표현된다.

e_1:*line_a-send(occupied)* e_2:*line_a-send(get_trunk)*
e_3:*line_a recv(return)* e_4:*line_a-recv(dialing_tone)*
e_5:*line_a-send(digit)* e_6:*line_b-recv(connect_l)*
e_7:*line_a-recv(free_tone)* e_8:*line_b-recv(ringing)*
e_9:*line_b-send(off_hook)* e_{10}:*caller-recv(tone_off)*
e_{11}:*line_b-recv(tone_off)* e_{12}:*line_a-send(voice_in)*
e_{13}:*line_b-recv(voice_out)* e_{14}:*line_b-send(on_hook)*
e_{15}:*line_a-send(on_hook)* e_{16}:*line_a-recv(literated)*
e_{17}:*line_b-recv(literated)* e_{18}:*line_a-send(unoccupied)*
e_{19}:*line_b-send(unoccupied)*

그림 10 이벤트들의 집합

두 개의 프로세스 그룹 M1 = {alloc_a, region, trunk, alloc_b}과 M2 = {caller, line_a, line_b} 사이의 상호작용을 테스팅 한다고 가정하자. 그러면, 우리는 그

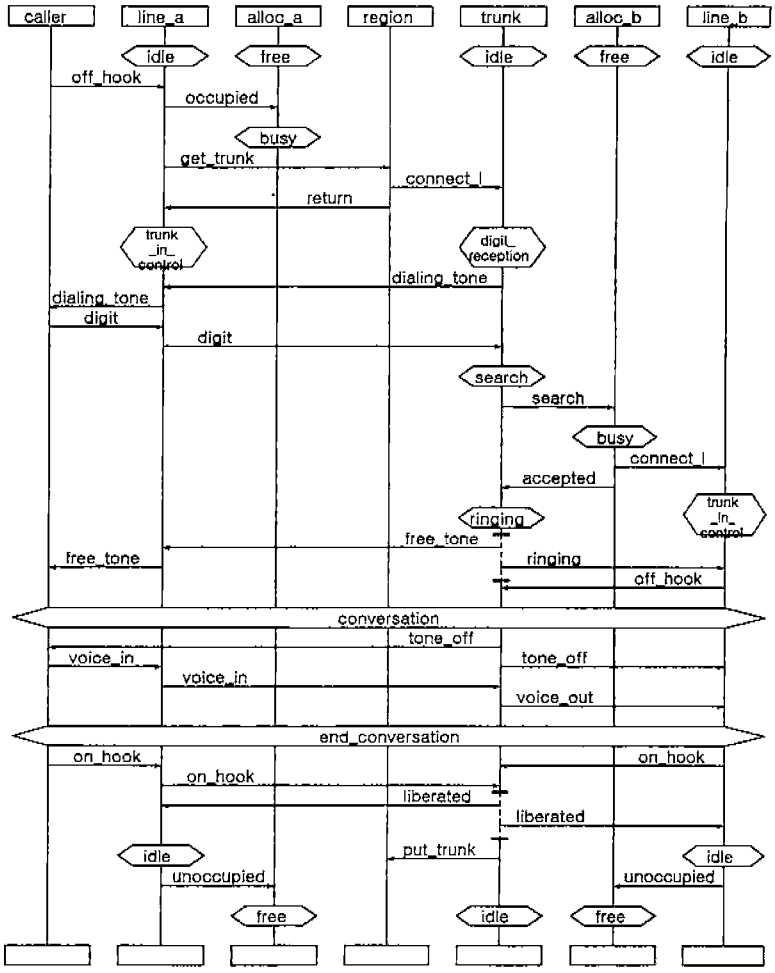

그림 9 전화 통화 시스템 MSC 명세

림 9의 가능한 모든 이벤트들로부터 두 프로세스 그룹간의 상호작용에 나타나지 않는 이벤트들(즉, 그룹 내부 프로세스간의 이벤트들)을 제거함으로써 그림 10과 같은 관심 있는 이벤트 집합 E_M을 구할 수 있다[7,8]. 이 시점에서 두 프로세스 그룹간의 상호작용을 테스팅하기 위해서 제어기가 필요한데 제어기는 두 개의 프로세스 그룹과 병행적으로 수행되면서 이들 사이의 상호작용을 제어하는 오토마타이다. 이벤트 시퀀스 ω와 이벤트들간의 종속관계 D가 주어지면, 앞 절에서 제시된 방법에 따라 그런 오토마타를 구축할 수 있다. 따라서, MSC 명세의 문맥상에서 임의의 순서로 수행될 수 없는 이벤트들인 종속 이벤트들을 파악하기 위한 조건들을 정의할 필요가 있다.

주어진 MSC M에 대해,

(조건 1) 만일 E_M에 속하는 이벤트 e_i와 e_j가 순차적이라면 즉, 전 순서 관계[2]라면, 두 이벤트 e_i와 e_j는 종속적이다.

(조건 2) 만일 E_M의 이벤트 e_i와 e_j가 서로 다른 프로세스에서 시작하는 메시지 전송 이벤트로서 병행관계이지만, e_i와 e_j에 대응되는 수신 이벤트 e_i'와 e_j'가 동일 프로세스 내에서 순차적으로 수신한다면, 두 이벤트 e_i와 e_j는 종속적이다.

2) 각각의 이벤트에 Fidge의 논리 시간표 벡터(logical vector time stamp)[19]를 붙이고 그것들 비교함으로써 MSC 명세에서 이벤트들 간의 순서 관계를 파악할 수 있다.

만일 이벤트 쌍 (e_i, e_j)가 위의 조건들을 하나도 만족하지 않는다면, 그것은 독립적이라고 간주된다. 위의 조건들은 MSC 명세의 제어 및 시그널 흐름을 기반으로 하고 있다. 그러므로, 그것들은 프로세스간에 변수들을 공유하지 않는 시스템에만 적용 가능하다. 우리는 자료 흐름 정보 또는 프로세스의 내부 구조에 관한 정보를 이용하여 조건들을 세분화할 수 있다. 그렇지만 이 논문에서는 시스템에 관한 추가적인 정보를 이용하여 종속관계를 세분화하기 위한 다양한 방법에 대해서는 다루지 않을 것이다. 조건 1과 2의 주요한 장점 중의 하나는 그것들은 정적으로 쉽게 검사될 수 있다는 것이다.

조건 1로부터, 전 순서 관계에 있는 임의의 두 이벤트들은 종속적이라 간주된다. 예를 들어, 이벤트 "line_a-send(occupied)"와 "line_a-send(get_trunk)"를 고려하자. 이벤트 "line_a-send(occupied)"가 이벤트 "line_a-send(get_trunk)"보다 항상 먼저 발생되어야 하기 때문에 그들은 종속적이라는 것을 알 수 있다. 조건 2를 고려하면, 메시지 수신의 순서가 프로세스 자신의 계산에 영향을 줄 때, 종속 관계가 발생함을 알 수 있다.

예를 들어, 그림 9의 "trunk" 프로세스는 "on_hook" 메시지를 "line_a"로부터 또한 "line_b"로부터 수신한다. 이것은 동일한 시스템에서 수행 중에 두 시나리오가 잠재적으로 발생할 수 있음을 보여준다.

물론 메시지 수신의 순서가 반드시 충돌을 유발하지는 않는다. 그러나, 경쟁 없음(race-free)이 보장되지 않는 비결정성 행위는 충돌의 적지 않은 원인이 된다. 이벤트 "line_b-send(on_hook)"과 "line_a-send(on_hook)"은 조건 2를 만족하므로 종속적임을 알 수 있다. 이와는 반대로, 이벤트 "line_a-recv(free_tone)"과 "line_b-recv (ringing)"은 비록 그들이 병행적이지만 위의 조건들 중 만족하는 게 없으므로 독립적이다. 사실, 이벤트 "line_b-send(on_hook)"과 "line_a-send(on_hook)"은 조건 2를 만족하는 유일한 이벤트 쌍이다. 즉, 병행적인 다른 이벤트 쌍들은 독립적이다. 이러한 고려를 바탕으로 우리는 이벤트 시퀀스 $\omega = e_1e_2\cdots e_ie_{i+1}\cdots e_{18}e_{19}$에 대한 오토마타를 그림 11과 같이 구축할 수 있다.

그림 11에서와 같이 일반적으로 테스트 제어기는 각각이 동기화 시퀀스를 나타내는 여러 개의 경로를 갖는다. 이 논문에서 제안한 테스팅 방법은 테스트 제어기의 여러 경로 중 적어도 하나가 MUT(Module Under Test)에 의해 실행되기를 요구한다. 반면 고전적인 결정적 수행 방법은 주어진 이벤트 시퀀스 $\omega = e_1e_2\cdots e_ie_{i+1}\cdots e_{18}e_{19}$를 나타내는 경로가 실행되기를 요구한다.

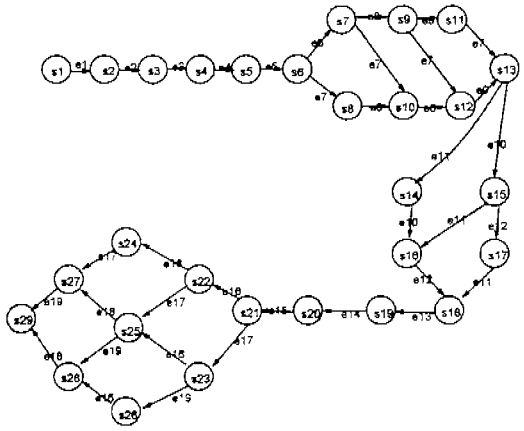

그림 11 구축된 오토마타

앞서 언급하였듯이, 명세에서 비결정성은 명세의 편리성을 위해 도입된다. 예를 들어, 그림 11의 두 비결정적 이벤트 "line_a-recv(free_tone)"과 "line_b-recv(ringing)"을 고려하자. 여기서는 피호출자가 "ringing" 시그널을 듣기에 앞서서 호출자가 "free_tone" 시그널을 듣던지 혹은 그 반대든 지가 문제가 되지 않는다. 이런 종류의 비결정성(선택적(optional) 비결정성[6] 혹은 영속적(persistent) 비결정성[20]이라 불린다.)은 결과에 영향을 주지 않고 설계 과정 중에 순차화될 수 있다. 가능한 시퀀스 중의 하나가 대응되는 프로그램에서 구현되었다고 가정하자. 그렇지만 그 시퀀스가 주어진 시퀀스와는 다르다고 한다면, 고전적인 결정적 수행 테스팅에서는 그 프로그램은 기대되는 행위를 보여주지 못하기 때문에 오류가 있다고 생각되어 진다.

5. 관련 연구

병행 프로그램 테스팅에서 중요한 관심사 중의 하나는 시스템이 정확하게 동작하는지 여부를 결정하는 것이다. [21]에서는 병행 프로그램의 시간 특성을 기술하고 검증하기 위해 그래픽 구간 논리(graphical interval logic: GIL)를 사용하고 있다. GIL에서 명세는 프로그램 수행 중에 실행되어야 할 모든 상태 시퀀스가 만족해야 하는 시간 특성을 기술한다. GIL의 형식으로 기술된 특성들이 만족되는지 여부를 결정하기 위하여, 그 형식을 만족하는 상태 시퀀스들을 받아들이는 유한 상태 기계(FSM)로 테스트 오라클을 구축한다. 이 접근 방법은 시퀀스들이 생성될 때 검사되고, 명세의 틀린 부분이 가능한 시퀀스의 초기에 보고되는 방법으로 실행 시간 감

시(monitoring)와 디버깅을 지원한다.

테스트 오라클 접근 방법의 주요 관심은 주어진 명세에 대해서 실행 시퀀스들을 검증하는 것이다. 따라서, 테스트 시퀀스를 생성하는 방법에 대해서는 많은 노력이 기울여지지 않았다. 이와는 반대로 이 논문에서 제시하는 방법은 테스트 시퀀스의 자동 생성에 초점을 맞추고 있다. 테스트 시퀀스는 프로그램 행위에 대해 주어진 시퀀스와 동일한 효과를 갖도록 생성된다. 생성된 시퀀스는 비록 그것이 GIL과 같은 정도로 프로그램의 특성에 대한 추론을 지원하지 않지만 MUT가 기대했던 대로 동작하는지 결정하기 위한 오라클로 사용될 수 있으며 오토마타로 표현된다.

테스트 오라클 접근 방법에 덧붙여, 명세를 기반으로 하는 두 타입의 병행 프로그램 테스팅 방법인 제약조건 중심의(constraints-oriented) 접근 방법과 트레이스 중심의 접근 방법을 고려할 수 있다. 제약조건 중심의 접근 방법에서는 개개의 제약조건에 대한 커버리지를 기반으로 테스트 기준이 정의된다[5]. 개개의 제약조건 (a, b)는 P의 수행 중에 두 이벤트 모두가 발생하면서 a가 b보다 선행한다면, P의 수행에 의해 커버된다고 말하여 진다. 비슷하게, 수행 중에 b가 a를 선행한다면 P의 수행은 제약조건 (a, b)를 위반한다고 말하여 진다. 주어진 이벤트 시퀀스의 실현 불가능성(infeasibility) 뿐만 아니라 제약조건의 커버리지를 보이기 위해서 고전적인 결정적 수행 테스팅을 사용할 수 있다.

비록 제약조건 중심의 접근 방법이 제약조건들을 하나 하나씩 조사하는데는 유용하지만, 그것은 명세와 프로그램간의 순응(conformance) 관계를 보장하지 않는다. 게다가, 제약조건 중심의 방법은 유효한 모든 시퀀스는 대응되는 프로그램에서 반드시 실현되어야 한다고 가정한다.

트레이스 중심의 접근 방법은 개개의 제약조건들 보다는 수행 트레이스 관점에서 명세와 프로그램간의 순응 관계를 보장하는데 주력한다. 트레이스 중심 테스팅 방법에 대한 대부분의 선행 연구들은 동치 순응 관계에 관심을 집중시키고 있다[4,13,14,15]. 이 관계를 기반으로 한 테스팅 기법들은 유효한 모든 시퀀스들은 실현 가능하고, 실현 가능한 모든 시퀀스들은 유효하다는 것을 보장함으로써 명세와 프로그램간의 동치 관계를 보이려 한다.

병행 프로그램 테스팅에 대한 또 다른 접근 방법은 프로토콜 테스팅으로부터 시작되었다. 프로토콜 테스팅으로부터 시작된 순응 테스팅은 보통 Estelle, Lotos, SDL과 같이 상태 기계 유사 모델로 쓰여진 명세를 기반으로 한다. 왜냐하면 통신 프로토콜은 반작용(reactive) 특성을 내포하고 있기 때문이다. 비록 현존하는 대부분의 프로토콜들이 완벽하게 명시되지 않지만, 전통적인 순응 테스팅 기법들은 주로 명세와 프로그램간의 동치 관계를 기반으로 한다. 따라서, 그들은 구현이 명세와 동일한 상태와 전이를 갖는다는 것을 보장하는데 주력한다[4,13].

6. 결 론

병행 프로그램 테스팅의 주요 관심사 중의 하나는 병행 프로그램의 비결정적 행위를 어떻게 제어하는 가이다. 이 사실로 인해 결정적 수행 테스팅이 높은 관심의 대상이 된 것이다. 왜냐하면 결정적 수행 기법은 동기화 시퀀스에 부합되는 경로를 따르도록 프로그램의 수행을 강제하기 때문이다. 그러나, 그것은 프로그램이 명세에 기술된 대로 정확하게 구현되지 않은 상황에서는 직접적으로 적용할 수 없다. 이 논문에서 비록 프로그램이 가능한 모든 유효한 시퀀스들의 부분 집합만을 실현했다 할지라도, 프로그램 행위에 대해 동일한 효과를 갖는 실현 가능한 대안 시퀀스가 존재한다면 프로그램은 그 명세를 만족할 수 있음을 알 수 있었다. 이러한 관찰은 결과에 영향을 주지 않는 비결정적 행위는 단지 병렬 수행의 속도를 향상시키기 위한 목적이지 본질적인 것은 아니라는 사실에 기초한다[20]. 그러나, 고전적인 결정적 수행 테스팅은 유효한 모든 시퀀스는 실현 가능할 뿐만 아니라 실현 가능한 모든 시퀀스는 유효하다는 것을 보증함으로써 명세와 프로그램간의 동치 관계를 보이려 하기 때문에 그러한 상황에 대해 적절한 수단을 제공하지 못한다.

주어진 시퀀스와 프로그램의 행위에 대해 동일한 효과를 갖는 동기화 시퀀스를 파악하기 위한 효과적인 방법으로 이벤트 독립의 개념을 사용하였다. 또한, 주어진 시퀀스와 동치 관계인 그런 시퀀스들을 받아들이는 오토마타를 구축하는 방법과 테스트 드라이버를 구축하기 위한 기초로서 그것을 사용하는 방법을 제시하였다. 우리의 접근 방법에 대한 활용을 보이기 위해 통신 소프트웨어 개발의 실제 예제를 사용하였다. 우리의 연구는 결정적 수행 테스팅의 취약한 면을 보완하는 방향으로 개선하였다. 향후 연구 방향은:

- ▶다양한 형태의 명세나 프로그램에 대해서 이벤트 독립을 파악하기 위한 분석 방법이 필요하다. Godefroid[5]는 LFCS(Labeled Formal Concurrent Systems)에 의해 표현된 병행 시스템에 대해 그런 분석이 어떻게 이루어 질 수 있는지를 설명했다.

▶우리의 방법을 지원하기 위한 소프트웨어 도구가 필요하다.

▶병행 프로그램을 테스팅하기 위한 테스트 슈트(suite)로서 동기화 시퀀스들 $T = \omega_1, \ldots, \omega_m$의 집합을 선택하기 위해 테스트 기준(criteria)을 정의할 필요가 있다. 테스트 슈트 T의 각 원소들은 $[\omega_1], \ldots, [\omega_m]$을 구축하는 기초를 제공할 것이다. 우리의 앞선 연구[7]에서는 MSCs와 같은 부분적이고 비결정적인 명세에 대한 몇 개의 테스트 기준을 제시했다.

참 고 문 헌

[1] K. C. Tai, "Testing Concurrent Programs," *9th Int'l Computer Software and Applications Conference: COMPSAC'85*, pp. 310-317, 1985.

[2] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Computers*, Vol. C-36, No. 4, pp. 471-482, 1987.

[3] K. C. Tai, R. H. Carver and E. E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. on Soft. Eng.*, Vol. 17, No. 1, pp. 45-63, 1991.

[4] G. V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," *Proc. of Int'l Symp. on Software Testing and Analysis*, pp. 109-124, 1994.

[5] R. H. Carver and K. C. Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. on Software Eng.*, Vol. 24, No. 6, pp. 471-490, 1998.

[6] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems-An Approach to the State-Explosion Problem*, PhD thesis, Univ. de Liege, 1994.

[7] 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," *Proc. of Int'l Symp. on Soft. Eng. for Parallel and Distributed Systems*, pp. 72-82, 1999.

[8] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and D. G. Lee, "Testing of Concurrent Programs After Specification Changes," *Proc. of Int'l Conf. on Software Maintenance*, pp. 199-208, 1999.

[9] D. Rosenblum, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52-61, 1991.

[10] H. W. Sohn, D. C. Kung, and P. Hsia, "State-based Reproducible Testing for CORBA Applications," *Proc. of Int'l Symp. on Soft. Eng. for Parallel and Distributed Systems*, pp. 24-35, 1999.

[11] 정인상, "프로그램 변환을 통한 JAVA 다중 스레드 프로그램의 결정적 테스팅", 한국정보과학회 논문지 B, 27권 6호, pp. 607-617, 2000.

[12] A. Mazurkiewicz. "Trace Theory, In Petri Nets: Applications and Relationships to Other Models of Concurrency," *Advances in Petri Nets*, 1986.

[13] H. AboElFotoh, O. Abou-Rabia, and H. Ural, "A Test Generation Algorithm for Systems Modelled as Nondeterministic FSMs," *IEE Software Eng. Journal*, pp. 184-188, 1993.

[14] S. K. Damodaran-Kamal and J. M. Francioni, "Testing Races in Parallel Programs with an OtOt Strategy," *Proc. of Int'l Symp. on Software Testing and Analysis*, pp. 216-227, 1994.

[15] G. H. Hwang, K. C. Tai, and T. L. Huang, "Reachability Testing : An Approach to Testing Concurrent Software," *Proc. of Asia Pacific Software Eng. Conf. 1994*, pp. 246-255, 1994.

[16] E. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. on Soft. Eng.*, pp. 56-66, May 1980.

[17] ITU-T Recommendation Z.120: Message Sequence Chart (MSC), April 1996.

[18] 김병만, 김현수, 신윤식, "MSC로 기술된 소프트웨어 명세의 검증을 위한 전체 상태 전이 그래프 생성", 한국정보과학회 논문지 B, 26권 12호, pp. 1428-1444, 1999.

[19] C. Fidge, "Logical Time in Distributed Computing Systems," *IEEE Computer*, Vol. 24, No. 8, pp. 28-33, 1991.

[20] N. Uchihira. S. Honiden, and T. Seki, "Hypersequential Programming: A New Way to Develop Concurrent Programs," *IEEE Concurrency*, pp. 44-54, 1997.

[21] L. K. Dillon and Q. Yu, "Oracles for Checking Temporal Properties of Concurrent Systems," *Proc. of 2nd ACM SIGSOFT Symp. on Foundations of Software Eng.*, pp. 140-153, 1994.

정 인 상

1983년 ~ 1987년 서울대학교 컴퓨터공학과(학사). 1987년 ~ 1989년 한국과학기술원 전산학과(석사). 1989년 ~ 1993년 한국과학기술원 전산학과(박사). 1994년 ~ 1998년 한림대학교 부교수. 1997년 ~ 1998년 미국 purdue 대학 방문교수. 1999년 ~ 현재 한성대학교 컴퓨터 공학부 부교수. 관심분야는 병렬 및 분산 프로그램 테스팅, 테스트 데이타 자동생성, 모형 검사(model checking)

김 병 만

1983년 ~ 1987년 서울대학교 컴퓨터공학과(학사). 1987년 ~ 1989년 한국과학기술원 전산학과(석사). 1989년 ~ 1992년 한국과학기술원 전산학과(박사). 1992년 ~ 현재 금오공과대학교 부교수. 1998년 ~ 1999년 미국 UC, Irvine 대학 방문교수. 관심분야는 프로그램 테스팅 및 검증, 인공지능, 정보검색

김 현 수

1988년 서울대학교 계산통계학과(학사). 1991년 한국과학기술원 전산학과(석사). 1995년 한국과학기술원 전산학과(박사). 1995년 3월 ~ 1995년 12월 한국전자통신연구원 박사후연수연구원. 1996년 3월 ~ 2001년 8월 금오공과대학교 컴퓨터공학부 조교수. 1999년 7월 ~ 2000년 7월 Colorado State Univ. 방문연구교수. 2001년 9월 ~ 현재 충남대학교 정보통신공학부 컴퓨터공학전공 조교수. 관심분야는 소프트웨어공학, 객체지향 소프트웨어공학, 소프트웨어 리엔지니어링, 분산객체 컴퓨팅, 컴포넌트기반 소프트웨어공학