MSC 명세를 기반으로 한 병렬 프로그램 테스팅 환경의 개발

(Development of a Testing Environment for Parallel Programs based on MSC Specifications)

김 현 수 [†] 배 현 섭 [††] 정 인 상 [†††] 권 용 래 [††††] 정 영 식 [†††††]

(Hyeon Soo Kim) (Hyun Seop Bae) (In Sang Chung) (Yong Rae Kwon) (Young Sik Chung)

이 병 선 [†††††] 이 동 길 [††††††]

(Byung Sun Lee) (Dong Gil Lee)

요 약 병렬 프로그램 테스팅을 위한 기존의 연구는 대부분 프로그램 수행 중에 얻어진 이벤트 트레이스를 바탕으로 재수행성을 보장하는데 중점을 두고 있다. 반면에 개발과정에서 만들어진 요구 명세로부터 테스트 케이스를 생성하는 방법에 대한 연구는 빈약한 실정이다. 본 연구에서는 통신 소프트웨어 개발 분야에서 광범위하게 사용되는 메시지 순차도(MSC)로 작성된 명세로부터 병렬 프로그램의 모듈 테스팅을 위한 테스트 케이스를 자동으로 생성하는 방법을 제안하고 생성된 테스트 케이스를 이용하여 실제 테스팅을 수행할 수 있는 환경을 개발하였다. 명세로부터 테스트 케이스를 자동으로 생성하기 위해서는 명세 내에 묵시적으로 포함되어 있는 이벤트들과 그들 간의 선후 관계를 파악해야 하는데 이를 위해서 본 연구에서는 논리시간벡터를 MSC 명세에 적용하기 위한 방법을 제안하여 이벤트간의 선후 관계인 이벤트 시퀀스를 추출하고 이를 테스트 케이스로 사용한다. 생성된 테스트 케이스는 TTCN 형태로 기술되고 이는 다시 CHILL 소스 코드 형태로 변환되어 테스트 대상이 되는 모듈과 상호 동작하면서 테스팅 대상 모듈의 동작이 기술된 요구 명세의 내용과 합치하는 지를 검사한다. 본 연구에서 개발한 테스팅 방법은 통신소프트웨어 개발 과정에서 산출된 MSC 명세로부터 테스트 케이스를 추출함으로써 테스팅을 위해 별도의 명세를 작성할 필요가 없다. 또한, 논리 시간 벡터를 적용하여 이벤트 시퀀스를 자동 생성할 뿐만 아니라 생성된 이벤트 시퀀스는 시스템 전체의 이벤트 시퀀스로써 독자적인 테스팅 방법으로 사용될 수 있다.

Abstract Most of prior works on testing parallel programs have concentrated on how to guarantee the reproducibility by employing event traces exercised during executions of a program. Consequently, little work has been done to generate test cases, especially, from specifications produced from software development process. In this research work, we devise the techniques for deriving test cases automatically from the specifications written in Message Sequence Charts(MSCs) which are widely used in telecommunication areas and develop the testing environment for performing module testing of parallel programs with derived test cases. For deriving test cases from MSCs, we have to uncover the causality relations among events embedded implicitly in MSCs. For this, we devise the methods for adapting vector time stamping to MSCs. Then, valid event sequences, satisfying the causality relations, are generated and these are used as test cases. The generated test cases, written in TTCN, are translated into CHILL source codes, which interact with a target module to be tested and test the validity of behaviors of the module. Since the testing method developed in this research work extracts test cases from the MSC specifications produced from telecommunications software development process, it is not necessary to describe auxiliary specifications for testing. In addition adapting vector time stamping generates automatically the event sequences, the generated event sequences that are ones for whole system can be used for individual testing purpose.

† 정 회 원 : 금오공과대학교 컴퓨터공학부 교수
hskim@cespc1.kumoh.ac.kr
†† 비 회 원 : 한국전자통신연구원 연구원
hsbae@bada1.etri.re.kr
††† 종신회원 : 한성대학교 정보전산학부 교수
insang@hansung.ac.kr
†††† 종신회원 : 한국과학기술원 전산학과 교수
kwon@salmosa.kaist.ac.kr
††††† 종신회원 : 한국전자통신연구원 연구원
yschung@etri.re.kr
bslee@etri.re.kr
†††††† 비 회 원 : 한국전자통신연구원 연구원
dglee@etri.re.kr
논문접수 : 1999년 1월 25일
심사완료 : 2000년 1월 6일

1. 서 론

교환기 소프트웨어는 지원하는 서비스의 광범위한 특성과 사회적인 중요성에 비추어 볼 때 반드시 고 수준의 신뢰성을 갖추고 있어야 하며 이를 위해서는 소프트웨어에 대한 효과적인 검증 방법이 필수적이다. 여러 가지 검증 기법들 중 실제 개발 환경에서 가장 효과적으로 사용할 수 있는 기법 가운데 하나는 프로그램에 대한 테스팅 방법이다.

교환기 소프트웨어는 교환기 자체가 가지는 본질적인 병렬성과 분산성 때문에 대부분 병렬 프로그래밍 언어로 작성된다. 병렬 프로그램에 대한 테스팅 기법은 근본적으로 병렬 프로그램 내에 포함되어 있는 비결정성을 고려해서 설계되어야 한다. 즉, 병렬 프로그램은 프로그램에 대해서 동일한 입력을 제공하고 반복 수행하였을 경우에 서로 다른 결과가 발생할 수 있기 때문에 기존의 순차 수행 프로그램에 대한 방법을 그대로 적용할 수 없다[1].

병렬 프로그램 테스팅에 관한 기존 연구에서는 테스트 케이스를 (입력 자료, 프로그램 수행 중에 발생하는 이벤트의 트레이스)로 정의하고, 프로그램의 재수행성을 보장하는데 중점을 두고 있다. 즉, 프로그램을 처음 수행했을 때 발생하는 이벤트 트레이스를 기록했다가 이를 이용해서 수행 결과를 이후에 재현해내는 방법에 대한 연구가 많이 진행되었다[1, 2]. 이런 방법은 재수행을 위한 이벤트 트레이스를 명세보다는 실제 프로그램의 수행으로부터 얻으므로 프로그램 기반 테스팅으로 분류될 수 있다.

한편 병렬 프로그램에 대한 명세 기반 테스팅에 대한 연구는 많이 진행되지 않았다. 명세는 병렬 프로그램이 수행 중에 만족해야 할 순서 제약조건(sequencing constraints)을 포함하고 있는데 이런 제약조건을 기반으로 이벤트 시퀀스를 생성할 수 있다. 명세를 S라 하고 S로부터 생성된 이벤트 시퀀스들의 집합을 ES(S)라 하면, 병렬 프로그램 P는 명세 S에 대해서 다음과 같은 두 가지 측면에서 검사될 수 있다[3]. 첫째, P의 정당성(validity)을 검사할 수 있다. P를 수행했을 때 발생하는 모든 이벤트 트레이스의 집합 R을 ES(S)와 비교해서 $R \subset ES(S)$이면 P는 S에 대해서 정당하다고 한다. 둘째, ES(S)의 실현성(feasibility)을 검사할 수 있다. ES(S) 내에 있는 각 이벤트 시퀀스들을 따르도록 프로그램 P를 강제 수행함으로써 명세에서 허용한 시퀀스들이 실제 프로그램에 구현되었는지를 검사할 수 있다.

본 연구에서는 메시지 순차도(Message Sequence Charts, MSC)로 작성된 병렬 프로그램의 요구 명세로부터 명세 기반 모듈 테스팅을 위한 테스트 케이스를 자동으로 생성하는 방법을 제안하고, 생성된 테스트 케이스를 이용해서 병렬 프로그램에 대한 테스팅을 수행하기 위한 테스팅 환경을 설계하고 구현하였다.

MSC로부터 테스트 케이스를 자동으로 생성하기 위해서는 명세 내에 묵시적으로 포함되어 있는 이벤트들 간의 순서 제약 조건을 추출하는 것이 매우 중요하다. 순서 제약 조건은 어떤 이벤트가 다른 이벤트보다 먼저 발생하는가를 결정한다. 병렬 프로그램의 수행 중에 발생하는 이벤트들 간에는 부분 순서 관계가 존재한다. MSC에 포함된 이벤트들 간의 이러한 부분 순서 관계를 추출하기 위해서 우리는 Fidge가 제시한 논리 시간 벡터(logical vector time stamp) 개념을 명세에 적용한다[4]. 논리 시간 벡터를 이용해서 MSC 명세 내에 있는 이벤트들간의 순서 관계를 검출하고 나서 이를 바탕으로 이벤트 시퀀스를 생성하며 생성된 이벤트 시퀀스는 프로그램의 정당성을 검사하기 위한 테스트 케이스로 사용된다.

생성된 테스트 케이스는 TTCN(Tree and Tabular Combined Notation)이라는 표준 표기법으로 기술되며 이는 계속해서 CHILL 소스 코드로 변환된다. CHILL 소스 코드는 테스트 케이스에 따라 이벤트를 발생시키고 테스트 데이타를 실어보내며 테스트 대상이 되는 모듈과 상호 동작한 이벤트 순서를 기록하는 테스트 스텁의 역할을 수행한다. 일반적으로 병렬 프로그램은 반복 수행할 때마다 다른 결과를 발생시킬 수 있기 때문에 병렬 프로그램을 테스팅하기 위해서는 여러 번의 반복 수행이 필요하다. 이를 위해 테스트 대상 프로그램과 테스트 스텁의 반복 수행을 제어하기 위한 테스트 드라이버 프로그램을 사용한다. 여러 번의 수행을 거치면서 테스트 결과는 저장되고 그 결과는 수행 후 분석 방법을 사용하여 테스팅 대상 모듈의 기능이 명세에 기술된 대로 바르게 동작하였는지를 검사하게 된다. 본 연구에서 개발된 여러 도구들은 개별적으로 구현되었지만 GUI를 사용한 환경으로 통합되어 테스팅 과정을 쉽게 따라 갈 수 있도록 구현되었다.

본 논문의 구성은 다음과 같다. 먼저 2절에서는 MSC에 대해서 간략하게 기술하고 3절에서는 테스팅 환경을 구성하고 있는 각 요소들에 대한 구체적인 설계 사항들을 기술한다. 4절에서는 각 부분의 구현에 대한 구체적인 사항들을 담고 있으며 5절에서는 관련 연구에 대해서 간략하게 검토하고 마지막으로 6절에서 결론을 맺고 향후 연구 방향을 설정한다.

2. MSC 명세 언어

MSC는 통신 시스템과 같은 분산/병렬 시스템에서 통신 실체들 간에 이루어지는 메시지 교환을 직관적으로 기술할 수 있는 그래픽 언어로서 ITU의 Z.120에 의해서 표준화되어 있다[5]. MSC는 초기에 단지 SDL (Specification and Description Language)의 부가적인 다이어그램 형태로 사용되어 오다가 점차 통신 시스템의 요구 명세를 기술하기 위한 정형적인 언어로 발전하게 되었으며 최근의 MSC 버전은 구조적 언어의 구조 및 객체 지향 언어의 개념 등을 포함하기에 이르렀다.

MSC는 기본적으로 시스템에 포함된 병렬 컴포넌트(프로세스)들과 주변 환경간의 메시지 교환을 표현하는데 중점을 두고 있다. MSC에서 제공하는 구성요소는 기본요소와 구조요소로 나뉠 수 있다. 기본요소는 프로세스들과 각각의 내부 행동, 또한 그들 간의 메시지 전달 등을 표현하기 위해서 사용하는 요소들로서 프로세스 인스턴스, 메시지, 환경, 내부 행동, 타이머, 프로세스 생성 및 소멸, 조건들과 병행영역(coregion)들로 구성된다. 구조요소는 다수의 메시지 순차도 간의 계층구조를 표현하기 위해서 사용되는 요소들이다. 자세한 내용은 권고안 Z.120[6]을 참조 바란다.

그림 2는 간단한 MSC 사용 예제로서 표준 MSC가 제공하는 대부분의 기본 요소들을 포함하고 있다. 그림에는 D-1, D-2, I-1, I-2의 네 개의 프로세스가 있는데 각각은 수직선으로 나타나있다. 프로세스를 가로지르는 육각형은 수행 중에 프로세스가 만족해야 하는 조건을 표현한다. 두 프로세스를 연결하는 수평 화살표들은 메시지 전달에 의한 통신을 의미한다. 마지막으로, 그림 2-(b)의 I-1 프로세스에는 수직 점선으로 표현된 부분이 나타나는데 이를 병행영역이라 부른다. 한 프로세스 내에 포함된 이벤트들은 수직선상의 위치에 따라 순서를 가지지만 병행영역에 포함된 이벤트들은 순서를 가지지 않는다. 예를 들어, I-1 프로세스는 순서 관계에 의해서 메시지 b를 보내기 전까지는 메시지 c를 보낼 순 없지만, 메시지 h와 메시지 i는 병행영역에 포함되어 있으므로 임의의 순서로 보낼 수 있다.

3. 테스팅 환경의 설계

병렬 프로그램 테스팅에서 테스트 케이스는 일반적으로 프로그램 수행 중에 발생되어야 하는 이벤트의 순차(sequence)로 표현된다. *이벤트*라는 용어는 구현 환경에 따라서 다른 의미로 사용되지만 근본적으로 시스템 내에 존재하는 독립된 개체들 간의 동기화 혹은 통신을 뜻한다. 본 논문에서는 메시지 전송을 기반으로 하는 병렬 프로그램에 대한 모듈 테스팅을 목적으로 하기 때문에 여기서의 이벤트는 프로세스들 간에 전달되는 메시지들을 의미한다.

병렬 프로그램 개발은 다양한 과정을 거치며 여러 가지 모델을 사용하지만 통신 분야에서 일반적으로 사용되는 방법은 MSC를 이용해서 프로그램의 요구 사항을 명세하고 SDL을 이용해서 설계 과정을 거친 후에 CHILL 언어로 프로그래밍 하는 과정이다. 이 때 SDL로부터 테스트 케이스를 생성할 수도 있고 CHILL 프로그램 자체에서 테스트 케이스를 생성할 수도 있지만 통신 프로그램에서 반드시 요구되는 테스팅 중의 하나는 요구 명세를 표현하고 있는 MSC와 실제 CHILL 프로그램간의 합치성을 검사하는 **기능 테스팅**(function testing)이다.

다음 그림 1은 본 연구에서 구축한 테스팅 환경의 전체적인 구조를 보여준다. 주요 도구들을 설명하면 다음과 같다.

- MSC 조합기 : 그래픽 에디터를 제공해서 MSC들간의 연속 관계를 표현할 수 있도록 하고 연속적인 여러 장의 MSC를 한 장의 MSC로 통합해준다.
- 테스트 케이스 생성기 : 통합된 MSC로부터 테스트 케이스를 생성하고 생성된 테스트 케이스를 TTCN 형태로 저장한다.

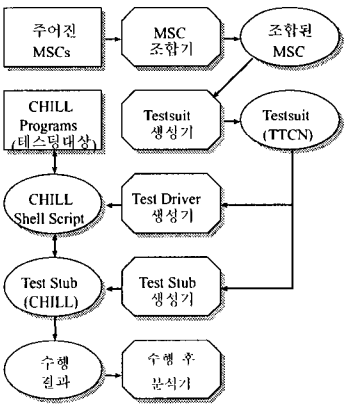

그림 1 MSC기반 테스팅 환경의 전체 구조

- 테스트 스텁 생성기 : 테스트 케이스 생성기에 의해서 산출된 TTCN 형태의 테스트 케이스로부터 테스트 스텁을 만든다. 만들어진 테스트 스텁은 CHILL 언어로 쓰여진 프로그램이며 이 프로그램은 MSC 내의 프로세스 그룹 중에서 테스팅 대상 이외의 그

룹이 담당해야 할 역할을 시뮬레이션 해주고 수행 중의 결과를 저장한다.

- 테스트 드라이버 생성기 : 테스팅 대상 프로그램과 테스트 스텁을 반복 수행시키고 그 수행을 조절한다.
- 수행 후 분석기 : 수행 후 테스트 스텁에 의해서 쓰여진 수행 결과를 분석한다.

3.1 MSC 조합

MSC는 병렬 프로그램에서 프로세스들 간의 상호 작용을 도식적으로 보여주기 때문에 요구 명세 단계에서 많이 사용된다. 요구 명세 언어로서 MSC는 두 가지의 큰 특징을 가진다. 첫째, MSC는 시스템의 시작부터 끝까지의 전체적인 행동을 모델링 하기보다는 관심 있는 부분만을 기술할 수 있다는 특징이 있다. 둘째, 한 장의 MSC는 특정한 스냅 샷만을 표현한다. 병렬 프로그램은 수행 경로 및 상호 작용에 따라서 무수히 많은 스냅 샷을 내포할 수 있는데 다른 모델들이 이런 많은 시나리오를 하나의 모델로 표현하려고 하는데 비해서 MSC의 경우에는 한번에 하나의 스냅 샷을 표현한다. 따라서 일반적으로 하나의 병렬 프로그램을 모델링하기 위해서 다수의 MSC가 그려진다.

이러한 MSC들간의 조합 방법은 두 가지 형태가 제안되어 있다. 그 중 하나는 최종 조건(final condition)과 초기 조건(initial condition)을 바탕으로 조합시키는 방법이고 다른 한 방법은 MSC'96에서 제안된 hMSC (high-level MSC)를 사용하여 MSC간의 조합을 기술하는 방법이다. hMSC를 사용하여 MSC간의 조합을 기술하는 방법은 [6]을 참고하기 바란다.

본 연구에서는 이 중에서 최종 조건과 초기 조건을 바탕으로 구성 MSC들을 조합하는 방법을 사용한다. 여기서 최종 조건이란 한 장의 MSC의 각 프로세스에서 해당 프로세스의 모든 이벤트들 이후에 나타나는 조건들을 의미하며, 초기 조건이란 각 프로세스에서 해당 프로세스의 모든 이벤트들보다 먼저 나타나는 조건들을 의미한다. 이 방법에서는 각각 특정 스냅 샷을 표현하는 두 개의 MSC (a)와 (b)가 존재한다고 가정할 때, 대응되는 프로세스 각각에 대해서 MSC (a)의 최종 조건과 MSC (b)의 초기 조건들이 동일한 이름이면 이들은 MSC (a)-(b)의 순서로 연속된 하나의 시나리오를 구성하는 것으로 취급된다. 예를 들어 그림 2의 MSC (a)와 (b)를 살펴보자. MSC (a)와 (b)는 D-1, D-2, I-1, I-2의 네 개의 프로세스로 구성되고, MSC (a)의 D-1의 최종 조건은 <busy>, D-2와 I-1의 최종 조건은 공유 조건인 <in_control>이고 MSC (b)의 D-1의 초기 조건은 <busy>, D-2와 I-1의 초기 조건은 공유 조건인 <in_control>이다. 이 경우에 MSC (a)의 최종 조건과 MSC (b)의 초기 조건들이 대응되는 각각의 프로세스에 대해서 동일하므로 MSC (a)의 시나리오가 먼저 진행되고 다음으로 MSC (b)의 시나리오가 진행되게 된다.

그러나 경우에 따라서는 그림 2의 MSC (a)의 프로세스 I-2와 같이 프로세스의 최종 조건이 나타나지 않을 수 있다. 이런 경우는 그 프로세스에서 어떤 조건이 나타난 이후로 그 상태가 계속 유지됨을 의미하며 이 경우에 후행 MSC의 대응되는 프로세스도 초기 조건을 갖지 않는다(그림 2의 MSC (b)의 프로세스 I-2 참조). 이런 경우에는 선행 MSC 내의 프로세스와 후행 MSC 내의 프로세스가 끊김 없이(상태 변화 없이) 수행되는 것으로 해석하여 조합한다. 그림 2의 MSC (a)와 (b)의 경우 이러한 조합 규칙에 의해 최종적으로 그림 3과 같이 조합된다.

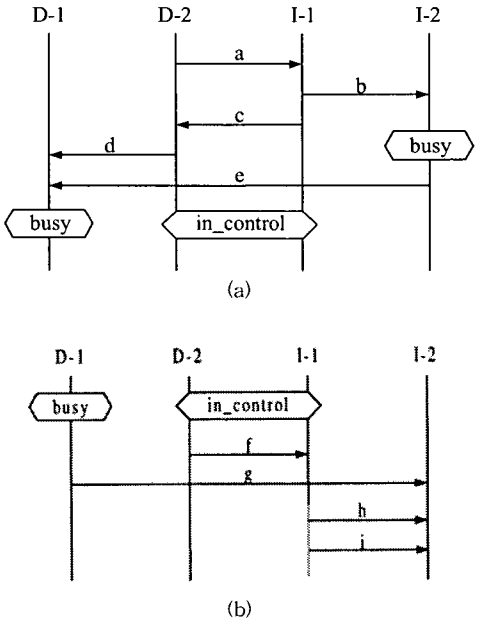

그림 2 예제 MSC

테스트 케이스를 생성하기 위해서는 현재 MSC 보다 먼저 생성되어야 하는 MSC를 반드시 고려하고 각 MSC들을 조합하여 처음부터 현재까지의 상황을 반영하고 있도록 해야한다. 예를 들어서 두 가입자가 전화를 하다가 끊었을 경우에 이 이벤트를 교환기가 제대로 처리하는지 검사하기 위해서는 먼저 두 가입자간에 전화가 연결되어 있어야 한다.

3.2 조합된 MSC로부터 테스트 케이스 생성

조합된 MSC로부터 테스트 케이스를 만드는 과정은 크게 다음과 같이 나눌 수 있다.

1. MSC 내의 프로세스들을 두개의 그룹으로 나눈다. 본 논문에서는 닫힌 MSC (closed MSC)를 가정한다. 즉, 모든 이벤트에 대해서 수신자와 송신자가 MSC 내에 포함된다. 따라서 일반적으로 MSC 내에 프로그램 외부 개체도 나타나게 된다. 또한 모듈 테스팅에서는 모든 모듈을 한꺼번에 테스팅하지 않고 한 모듈씩 따로 테스팅 하기 때문에 MSC 내의 일부분의 프로세스에 대해서만 테스팅 해야한다. 따라서 테스팅 대상이 되는 프로세스와 그렇지 않은 프로세스들을 묶어줘야 한다. 이때 테스트 케이스는 테스팅 대상 프로세스 그룹 I(IUT: Implementation Under Testing)와 그 외 프로세스 그룹 D 간의 이벤트 시퀀스가 된다.
2. 각 이벤트들에 대해서 논리 시간표(logical time stamps)를 부여한다. I에서 D로의 메시지들과 D에서 I로의 메시지들은 각자 내부 수행에 따라서 순차적으로 혹은 병렬 적으로 전송된다. 이때 이벤트들 간의 순서 관계를 파악하기 위해서는 각 이벤트들 간의 부분 순서 관계를 파악하고 이 순서를 보존해야 한다. 이를 위해서 본 논문에서는 Fidge의 논리 시간표를 각 이벤트에 부여한다. 각 이벤트에 논리 시간표를 부여하는 규칙은 3.2.1절에서 다룬다.
3. 각 그룹 내의 내부 이벤트들을 삭제한다. 테스팅에서 필요한 것은 두 그룹간의 상호 작용이 올바르게 발생하는지 여부이므로 각 그룹의 내부 이벤트들은 테스트 케이스에 포함되지 않아야 한다. 따라서 테스트 케이스 생성 전에 내부 이벤트들을 제거한다.
4. 두 프로세스 그룹간에 발생 가능한 이벤트 시퀀스를 생성한다. 각 이벤트에 할당된 논리 시간표에 위배되지 않도록 이벤트 시퀀스를 생성한다. 한 MSC는 하나의 스냅 샷을 나타내지만 그 스냅 샷은 일반적으로 부분 순서만을 내포하기 때문에 하나의 MSC 에서 여러 개의 시나리오가 발생할 수 있다. 이벤트 시퀀스 생성 규칙은 3.2.2절에서 다룬다.

그림 3은 조합된 간단한 MSC와 그에 대한 벡터 시간표를 보여준다. 그림 3의 MSC에서 내부 이벤트를 삭제한 결과가 그림 4에 나타나 있다.

이상과 같은 과정을 거쳐서 테스트 케이스를 만들 수 있다. 그림 4에 있는 축약된 MSC에 대해서 위 과정을 적용해서 테스트 케이스를 생성하면 여섯 개의 테스트 케이스가 생성되는데 그림 5는 그 일부분을 보여주고 있다.

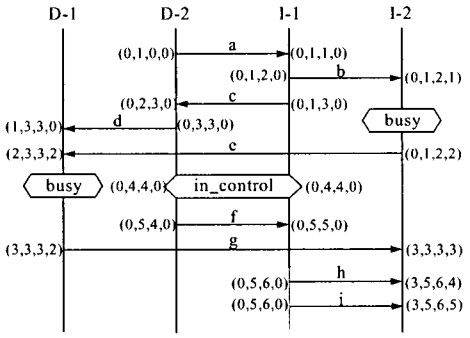

그림 3 MSC와 벡터 시간표

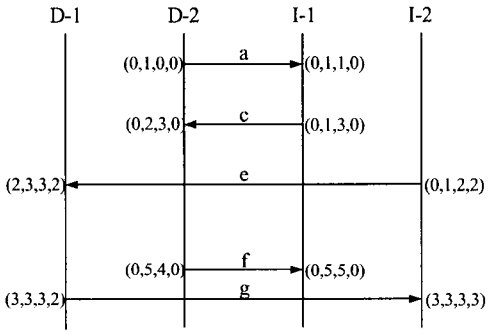

그림 4 내부 이벤트가 삭제된 MSC

TestSuitName : *example*-testsuit
Purpose : I = {*I-1, I-2*}, D = {*D-1, D-2*}
Comments : < >

TestCaseName : *example[1]*

Number	BehaviorDescription	Constraints	Verdict	Comments
1	B:a to C	<T1,DefaultInitializeT1( )>		A = *D-1*
2	B?c from C			B = *D-2*
3	A?e from D			C = *I-1*
4	B:f to C	<T2, DefaultInitializeT2( )>		D = *I-2*
5	A:g to D	<T3, DefaultInitializeT3( )>		

End TestCase

TestCaseName : *example[2]*

Number	BehaviorDescription	Constraints	Verdict	Comments
1	B:a to C	<T1,DefaultInitializeT1( )>		A = *D-1*
2	B?c from C			B = *D-2*
3	A?e from D			C = *I-1*
4	A:g to D	<T3, DefaultInitializeT3( )>		D = *I-2*
5	B:f to C	<T2, DefaultInitializeT2( )>		

End TestCase
....
End TestSuit

그림 5 TTCN으로 기술된 테스트 케이스

3.2.1 논리 시간 부여 규칙

이벤트에 논리 시간 벡터를 할당하는 방법은 MSC의

기본 구성요소에 따라서 다섯 가지의 규칙으로 설명될 수 있다. 이 규칙들을 설명하기 전에 이벤트들을 구분하기 위해서 다음과 같이 각 이벤트에 이름을 붙이는 방법을 생각하자.

- **RECV(x, p_i)**는 프로세스 p_i에서 메시지 x를 받는 이벤트를 뜻한다.
- **SEND(x, p_i)**는 프로세스 p_i에서 메시지 x를 보내는 이벤트를 뜻한다.

MSC는 프로세스와 그들 간의 메시지 전송 및 공유 조건, 그리고 병행 영역으로 구성된다. 따라서 MSC 내에 포함된 이벤트들 간의 순서관계를 파악하려면 MSC의 구성요소인 프로세스, 메시지 송/수신 이벤트, 공유 조건 등에 대해서 논리 시간 벡터를 할당해야 한다. 다음의 규칙 1과 5는 각 프로세스에 대한 논리 시간 벡터 할당 규칙이며 규칙 2는 한 프로세스 내에 포함된 이벤트들 간의 선후 관계를 표현하기 위한 규칙이다. 또 규칙 3은 메시지 송/수신 이벤트간의 선후 관계를 나타내고 규칙 4는 공유 조건에 의한 선후 관계를 표현한다.

- **규칙 1 : 초기화 규칙**

각 프로세스는 초기에 영벡터를 논리시간 벡터로 가진다. 즉, 프로세스 p_i의 현재 논리시간 벡터를 T_i라 하면 초기에 T_i=(0, ..., 0)이다. 벡터 T_i의 차원(dimension)은 주어진 MSC 명세에 포함되어 있는 프로세스 개수와 같다. 예를 들어, 그림 3에는 네 개의 프로세스가 존재하므로 모든 이벤트들은 4차원의 시간 벡터를 가진다.

- **규칙 2 : 상속 규칙**

기본적으로 각 이벤트는 자신이 속한 프로세스 내의 선행 이벤트로부터 시간 벡터를 상속 받는다. 이 규칙은 이벤트가 병행영역에 있는지 여부에 의해서 영향받는다. T_{ij}를 프로세스 p_i의 j번째 이벤트가 가지는 시간 벡터라 하자.

▶ 규칙 2-1 : 정상 상속

프로세스 p_i의 j번째 이벤트와 j+1번째 이벤트가 둘 다 병행영역에 있지 않은 경우에 ''T_{ij+1} = T_{ij} + i번째 단위벡터''이다. 다시 말해서, 병행영역에 있지 않은 이벤트는 자신의 선행 이벤트로부터 시간 벡터를 상속받은 후 i번째 행의 값을 하나 증가시켜서 자신의 시간 벡터를 만든다.

▶ 규칙 2-2 : 병행영역 상속

프로세스 p_i의 j+1번째, ..., m번째 이벤트가 병행영역에 있는 이벤트라고 가정하고 j번째 이벤트가 병행영역에 포함되지 않은 마지막 이벤트라고 하자. 그러면 병행영역에 포함된 이벤트들의 시간벡터는 '' $\forall x \in$ {j+1, ..., m}, T_{ix} = T_{ij} + i번째 단위벡터''이다. 즉, 병행영역 내의 모든 이벤트들은 동일한 논리시간벡터를 가지는데 이 벡터는 병행영역 직전의 이벤트로부터 시간벡터를 상속받은 후 i번째 행의 값을 하나 증가시켜서 만든다. 그림 3의 I-1 프로세스 내에 있는 병행영역을 고려해보자. 두개의 메시지 송신 이벤트 h와 i는 동일한 시간 벡터 (0,5,6,0)을 가지며 이는 f 이벤트로부터 상속받은 것이다.

- **규칙 3 : 통신 규칙**

두 개의 프로세스가 메시지를 주고받는 경우에 그들 간에는 묵시적인 선후관계가 형성되며 이 선후관계는 다음과 같은 규칙에 의해서 표현될 수 있다.

▶ 규칙 3-1 : 송신 규칙

프로세스 p_i의 j번째 이벤트가 송신 이벤트인 경우에 이 이벤트의 시간 벡터는 규칙 2를 이용해서 계산된 후 수신 프로세스 쪽으로 전달된다.

▶ 규칙 3-2 : 수신 규칙

프로세스 p_m의 n번째 이벤트가 수신 이벤트인 경우에 T_{mn}은 규칙 2를 통해서 계산된 자기 자신의 시간 벡터와 규칙 3-1을 통해서 송신 프로세스로부터 전달받은 시간 벡터를 행별로 최대값을 취함으로써 만들어진다.

예를 들어, 그림 3의 프로세스 D-2에서 메시지 c를 수신하는 이벤트를 고려해보자. 이 이벤트의 시간 벡터 (0,2,3,0)은 PAIRWISE_MAX{(0,2,0,0),(0,1,3,0)}을 통해서 만들어진다. 이때 (0,2,0,0)은 SEND(a, D-2) 이벤트로부터 상속받은 시간 벡터이고 (0,1,3,0)은 SEND(c, I-1) 이벤트로부터 전달받은 시간 벡터이다.

이상과 같은 기본요소 외에 시스템의 상태를 표현하기 위해서 사용되는 조건문과 프로세스 생성 및 소멸에 대한 고려가 필요하다. MSC에서 사용되는 조건문은 한 프로세스의 상태를 표현하기 위한 지역 조건문과 두개 이상의 프로세스들이 동시에 만족하는 상태를 표현하기 위한 공유 조건문으로 나뉜다. 특히 공유 조건문은 프로세스들 간의 묵시적 동기화를 표현하게 되므로 이를 논리시간 벡터에 반영해야 한다. 또한 MSC에서는 프로세스의 생성과 소멸을 표현할 수 있다. 프로세스의 동적 생성이나 소멸이 허용되는 경우에 프로세스의 수가 가변적이므로 시간 벡터의 차원을 결정하기가 어렵게된다. 이 문제는 ''순서쌍의 집합''과 같은 자료 구조를 통해서 해결할 수 있으며 여기서는 문제를 용이하게 하기 위해서 프로세스의 개수는 미리 알려져 있다고 가정한다. 이 경우에 프로세스 소멸에 대한 별도의 규칙은 필요치 않으며 다만 프로세스 생성을 처리할 수 있는 규칙이 필요하다. 다음의 규칙 4와 규칙 5는 각각 공유 조

건문과 프로세스 생성을 처리하기 위한 규칙들이다.

• **규칙 4 : 공유 조건 규칙**

하나의 조건문을 공유하는 프로세스 (p_j, ..., p_l)들은 각 시간 벡터들의 최대값을 각자의 시간 벡터로 가진다. 즉, ``$\forall x \in \{j, ..., l\}$, T_{xc} = PAIRWISE_MAX{T_j, ..., T_l}''이며 여기서 T_{xc}는 공유 조건문에 대한 시간 벡터이다. 예를 들어, 공유 조건 in_control의 시간 벡터 (0,4,4,0)은 PAIRWISE_MAX{(0,4,3,0),(0,1,4,0)}으로부터 계산되어진다.

• **규칙 5 :프로세스 생성 규칙**

프로세스 p_i가 프로세스 p_j를 생성하는 경우에 p_j는 p_i의 현재 시간 벡터를 취한다. 즉, ``$T_j = T_i$''이다.

이상과 같은 시간벡터 할당 규칙에 덧붙여서 이벤트들 간의 선후관계를 비교하기 위한 비교 규칙이 필요하다. 이벤트들 간의 선후관계는 '$\rightarrow$'로 표현되며 다음의 규칙을 이용해서 판단될 수 있다.

• **비교 규칙**

프로세스 p_i 내의 이벤트 a와 프로세스 p_j 내의 이벤트 b 간의 선후 관계는 다음과 같이 결정된다: $a \rightarrow b \Leftrightarrow T_{ia}(i) \leq T_{jb}(i) \wedge T_{ia}(j) < T_{jb}(j)$ 여기서 T(k)는 시간 벡터 T의 k번째 행의 값을 의미한다.

3.2.2 이벤트 시퀀스 생성 규칙

내부 이벤트가 삭제된 MSC로부터 이벤트 시퀀스를 생성하는 기본 규칙은 논리 시간 벡터에 나타난 부분 순서관계를 만족하는 모든 교차 시퀀스(interleaving sequence)를 생성하는 것이다. 교차 규칙은 프로세스 그룹 I와 D간의 통신 패턴에 따라서 달라진다.

프로세스 그룹 G 내의 모든 이벤트들의 집합을 E(G)라 하자. 여기서 $G \in \{I, D\}$이다. 내부 이벤트가 삭제된 상태이기 때문에 RECV(x, p_i) $\in$ E(D)이면 SEND(x, p_j) $\in$ E(I)이고 그 역도 성립한다. 임의의 이벤트 $e \in E(G)$에 대해서 대응 이벤트는 e^o로 표현한다. 이벤트 집합 E(G)는 송신 이벤트들만을 포함하는 집합 '![E(G)]'와 수신 이벤트들만을 포함하는 집합 '?[E(G)]'의 두개의 부분집합으로 분할될 수 있다. 이상과 같은 기호를 이용해서 다음의 규칙들은 프로세스 그룹 D가 프로세스 그룹 I를 시뮬레이션 한다는 관점에서 교차 규칙을 기술한다.

• **생성 규칙 1 : 병행 송신**

두 이벤트 $e_1, e_2 \in$![E(D)]에 대해서 $e_1 \not\rightarrow e_2$이고 $e_2 \not\rightarrow e_1$이면 e_1과 e_2는 두 개의 이벤트 시퀀스로 교차된다.

• **생성 규칙 2 : 병행 수신**

두 이벤트 $e_1, e_2 \in$?[E(D)]에 대해서 $e_1 \not\rightarrow e_2$이고 $e_2 \not\rightarrow e_1$이면 e_1과 e_2는 두 개의 이벤트 시퀀스로 교차된다.

• **생성 규칙 3 : 순차 송신**

두 이벤트 $e_1, e_2 \in$![E(D)]에 대해서 $e_1 \rightarrow e_2$이면 e_1과 e_2는 하나의 이벤트 시퀀스로 순차화 된다.

• **생성 규칙 4 : 병행 송신으로부터의 순차 수신**

두 이벤트 $e_1, e_2 \in$?[E(D)]에 대해서 $e_1 \rightarrow e_2$이고 두 이벤트 e_1'와 e_2'간에 순서관계가 없으면 (즉, $e_1' \not\rightarrow e_2'$이고 $e_2' \not\rightarrow e_1'$), e_1과 e_2는 두 개의 이벤트 시퀀스로 교차된다.

• **생성 규칙 5 : 순차 송신으로부터의 순차 수신**

두 이벤트 $e_1, e_2 \in$?[E(D)]에 대해서 $e_1 \rightarrow e_2$이고 $e_1' \rightarrow e_2'$이면 e_1과 e_2는 하나의 이벤트 시퀀스로 순차화 된다.

• **생성 규칙 6 : 병행 송/수신**

두 이벤트 $e_1 \in$![E(D)]와 $e_2 \in$?[E(D)]에 대해서 e_1과 e_2 사이에 순서관계가 없으면 e_1과 e_2는 두 개의 이벤트 시퀀스로 교차된다.

• **생성 규칙 7 : 순차 송/수신**

두 이벤트 $e_1 \in$![E(D)]와 $e_2 \in$?[E(D)]에 대해서 $e_1 \rightarrow e_2$이면 e_1과 e_2는 하나의 이벤트 시퀀스로 순차화 된다. 마찬가지로 $e_2 \in$![E(D)]와 $e_1 \in$?[E(D)]에 대해서 $e_2 \rightarrow e_1$이면 e_2와 e_1은 하나의 이벤트 시퀀스로 순차화 된다.

3.3 테스트 스텁 생성

테스트 스텁은 CHILL 언어로 작성된 프로그램으로서 MSC 내의 프로세스 그룹 중에서 테스팅 대상 이외의 그룹이 담당해야 할 역할을 시뮬레이션 해주고 수행 중의 결과를 저장한다. 다시 말해, 테스팅 대상이 되는 IUT 프로세스들의 기능이 정확하게 동작하는 가를 테스팅하기 위해 테스팅 대상 그룹에 적절히 메시지를 전송하고 수신하는 기능을 수행한다. 이때, IUT 프로세스 그룹과 그 이외의 프로세스 그룹과의 메시지를 통한 통신 순서를 시나리오라 부르며, 이는 앞 절에서 추출된 테스트케이스를 의미한다.

IUT 프로세스들의 기능을 테스팅하기 위해서는 전송되는 메시지에 데이타를 실어 보내야 하는 경우와 수신된 메시지가 데이타를 포함하고 있어야 하는 경우가 있다. 전자의 경우에는 실어보낼 데이타를 결정해야 하고 후자의 경우에는 받은 데이타의 값을 점검해야 한다. 구체적이고 다양한 데이타 값을 가지고 테스팅 하는 것은

데이타 의존성 분석 및 테스팅 기법을 필요로 한다. 그러나 현재 MSC에는 데이타 흐름이나 데이타 의존 관계 혹은 데이타에 대한 제약 사항 등을 표현할 수 있는 방법이 제공되지 않고 있는 실정이다. 따라서 MSC 기반 테스팅 환경에서 순수 MSC를 이용해서 테스트 케이스를 생성하는 경우에 데이타 값을 적절하게 생성해 내는 것은 사실상 불가능하다. 이런 문제를 해결하기 위해서 본 환경에서는 사용자가 다양한 방식으로 데이타를 지정할 수 있도록 하였다. 본 환경에서 지원하는 데이타 지정 방식은 다음과 같다.

- GUI 환경에서 대화 상자를 통해서 데이타 값을 직접 지정할 수 있다.
- 화일에 데이타를 저장해 놓고 화일에서 데이타를 읽어 들이도록 지정할 수 있다.

테스트 스텝을 생성하기 위한 작업은 TTCN으로 기술된 테스트 케이스를 입력으로 받아 들여 파싱한 다음 그 결과를 AST(Abstract Syntax Tree)로 구축하고, 구축된 AST의 각 노드를 차례로 방문하면서 각 노드의 내용을 적절한 CHILL 코드로 변환하는 과정으로 이루어진다. 변환 규칙은 3.3.1절에서 다룬다. TTCN으로 기술된 테스트 케이스의 형태는 그림 5와 같으며 이에 대한 AST의 구조는 그림 6과 같다.

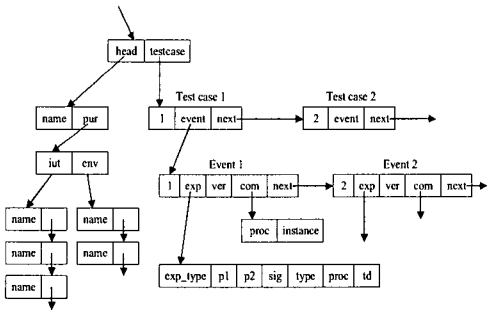

그림 6 테스트 슈트에 대한 AST의 구조

3.3.1 TTCN으로부터 CHILL 코드로의 변환

- **메시지 선언**

AST를 방문하면서 나타나는 메시지들을 모두 SEIZE 선언한다. 예를 들어, P1 !off_hook to Pabx의 경우 **SEIZE off_hook;** 문을 생성한다.

- **프로세스 선언**

AST를 방문하면서 **to**나 **from** 뒤에 나타나는 프로세스들에 대해 각각들을 타입이 INSTANCE mode인 변수(variable)로 선언하고 NULL 값으로 초기화한다. 예를 들어, P1 !off_hook to Pabx의 경우 **DCL Pabx INSTANCE := NULL;** 문을 생성한다.

- **최초 수신 메시지 선언**

프로세스 P가 다른 어떤 메시지를 송신(SEND) 하기에 앞서서 메시지 sig_a를 수신(RECEIVE) 할 경우: 프로세스 P가 어떤 메시지도 송신하지 않았기 때문에 P로 메시지를 송신하려는 다른 프로세스에서는 P의 pid(process id)를 알지 못한다. 따라서 송신할 메시지를 방송(broadcasting)하게 된다. 이 경우에 그 메시지를 받기 위해서는 그 메시지에 대해 최초 수신 메시지 선언을 해주어야 한다. 즉, **link(_sig_a, THIS);** 문을 생성한다.

- **메시지 송신문 변환**

AST를 방문하면서 메시지 송신을 만났을 때(예를 들어, P1 !digit to Pabx의 경우)

1) 만일 인스턴스 변수 Pabx = NULL이면, **SEND digit;** 문을 생성한다. 이 경우는 Pabx의 pid를 알지 못하므로 메시지 digit를 방송하게 된다.
2) 그렇지 않으면, **SEND digit TO Pabx;** 문을 생성한다.
3) 메시지가 데이타를 갖고 있으면 데이타 초기화 루틴을 통해 데이타를 초기화한 후에 전송한다. **Init_msg(m1); SEND digit(m1) TO Pabx;** 문을 생성한다. 여기서 m1은 전송될 데이타 변수이고 Init_msg는 전송될 데이타의 초기화 루틴이다.

- **메시지 수신 변환**

AST를 방문하면서 메시지 수신을 만났을 때(예를 들어, P1 ?off_hook from Pabx의 경우)

1) **RECEIVE CASE** 문을 생성한다. 이때 case 문의 항목에는 현재 테스팅 되는 시스템의 모든 메시지가 나열된다. 이렇게 하는 이유는 기본적으로 프로세스들이 병렬적으로 동작하기 때문에 기대하던 메시지가 도착하지 않고 다른 메시지가 먼저 도착할 경우에도 프로세스의 수행을 계속 진행시키기 위함이다.
2) **RECEIVE CASE** 문 내의 case 항목 중 현재 기대하는 메시지(예, off_hook)의 경우 메시지가 전송 데이타를 갖고 있으면 데이타 검사 루틴을 통해 검사한 후, 필요한 동작을 수행하고, 기대하던 메시지를 받았음을 기록하고, 최종적으로 타이머를 삭제한다.
3) **RECEIVE CASE** 문 내의 case 항목 중 현재 기대하는 메시지가 아닌 경우 도착한 메시지에 대

해 기록하고 타이머를 삭제한다.

4) **예외 처리** : 기대하던 혹은 기대하지 않은 메시지가 도착한 경우에는 계속 진행되지만 현재 메시지를 기다리고 있는 시점에서 어떤 원인에 의해 아무 메시지도 도착하지 않게 되는 상황이 발생하면 테스트 스텁은 계속해서 메시지를 기다리게 되고 테스팅 과정은 더 이상 진행될 수 없다. 이런 상황에 대비하여 타이머를 **RECEIVE CASE** 문에 앞서 지정하여야 한다. 만일 타이머 지정 시 설정한 시간 내에 아무런 메시지도 도착하지 않는다면 타이머는 time out을 발생시키고 지정해 놓은 메시지를 테스트 스텁에게 전송한다. 테스트 스텁은 이 메시지를 수신하면 그것을 기록한 후 테스트 스텁의 수행을 종료시킨다. 위의 모든 경우를 고려할 때 메시지 수신을 위해 생성되는 문장은 다음과 같다.

```
tid := set_timesig(10, SEC, _TS0);    /* 예외 처리를 위해
timer 등록 */
RECEIVE CASE NONPERSISTENT SET Pabx;
        (off_hook) : Check_msg(m2);
                action; logging; cancel(tid);
        (other signal-1) : logging; cancel(tid);
        (other signal-2) : logging; cancel(tid);
                ....
                (TS0) : logging;
                    STOP;
        ESAC;
```

- **프로세스 생성 및 소멸**

AST를 방문하면서 Create Process, Stop Process를 만났을 때(예를 들어, **P1 !create C,** 혹은 **stop C** 의 경우) : create에 의해 생성된 프로세스는 원래부터 테스트 스텁 프로세스 그룹에 존재하던 프로세스로 고려한다. 결국 테스트 스텁 프로세스 그룹은 하나의 테스트 스텁 프로세스 인스턴스로 표현되므로 생성된 프로세스의 메시지 송신 및 수신의 경우는 위의 일반적인 프로세스의 메시지 송신 및 수신의 경우와 동일하게 처리하면 된다.

- **Timer 관련 변환**

AST를 방문하면서 Timer와 관련된 요소를 만났을 때(**set timer, timeout, reset**의 경우)

1) Timer 이름을 INT 타입의 변수로 선언, timer가 사용하는 메시지에 대해 정의하고 선언

2) Set Timer의 경우 : **Timer := set_timesig (time_delay, SEC, _TSn);** 와 같은 timer 등록 문을 생성한다.

3) TimeOut의 경우 : timer에 등록된 timer 메시지가 도착하면 정상, 다른 메시지가 도착하면 비정상으로 간주한다.

```
RECEIVE CASE NONPERSISTENT
  (other signal-1) : logging; cancel(Timer); /* 비정상적인 경
  우 */
  (other signal-2) : logging; cancel(Timer); /* 비정상적인 경
  우 */
  ....
  (TSn) : logging;  /* Time out이 발생(정상) 기대하던 메시지
  = TSn */
 ESAC;
```

4) Reset의 경우 : 기대하던 메시지를 수신하면 timer를 reset한다. 기대하던 메시지를 수신하지 못하면 위의 일반적인 메시지 수신의 경우와 같이 예외 처리를 수행한다.

```
Timer := set_timesig(time_delay, SEC, _TSn);  /*timer 등록*/
.....
tid := set_timesig(10, SEC, _TS0);    /* 예외 처리를 위한
    timer 등록 */
RECEIVE CASE NONPERSISTENT SET Pabx;
    (expected signal) : action; cancel(tid);
                        cancel(Timer);
    (other signal-1) : logging; cancel(tid);
    (other signal-2) : logging; cancel(tid);
    ....
    (TS0) : logging;
        STOP;
    ESAC;
```

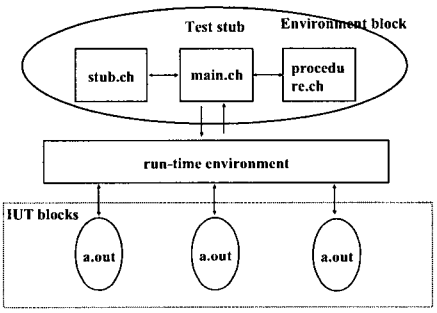

그림 7 테스팅 수행 모델

3.3.2 테스트 스텁

테스팅을 위한 시스템의 수행 모델은 다음과 같다. 여기서 테스트 스텁은 테스팅 대상이 되는 IUT 프로세스 그룹을 제외한 나머지 모든 프로세스들을 하나의 프로세스로 통합하여 테스팅 대상이 되는 IUT 블록들을 테스팅하기 때문에 IUT 입장에서 볼 때 테스팅 환경에

대응되므로 환경 블록(environment block)이라고도 부른다. Run-time environment는 CHILL 프로그램을 수행시킬 수 있는 환경을 의미하며 IUT 블록 내의 여러 개의 a.out은 IUT 블록에 속하는 각각의 프로세스들에 대한 실행 코드들이다.

테스팅 환경 블록은 세 개의 CHILL 프로그램으로 구성되며 각각은 main 모듈, 여러 함수들을 모아 놓은 procedure 모듈, 그리고 테스트 데이타의 입력과 수신된 데이타를 검사하는 함수들을 모아놓은 stub 모듈이 있다.

실제로 각각의 시나리오(하나의 시나리오는 하나의 테스트 케이스에 해당한다.)에 대해 별도의 테스트 스텁을 생성하는 방법도 있을 수 있지만 사실, 각각의 시나리오는 대부분이 동일하고 단지 그 순서만이 시나리오마다 다르기 때문에 시나리오마다 하나씩의 테스트 스텁을 생성하는 방법은 많은 코드의 중복을 가져온다. 이러한 문제점을 해결하는 방법으로 본 연구에서는 공통적으로 사용되는 함수들을 procedure 모듈에 모아 두고 main 모듈에서 그것들을 불러 쓰는 구조를 취하고 있다.

Main 모듈은 특정 시나리오에 맞춰 테스팅을 진행해 가는 모듈이다. 이 모듈의 구조는 그림 8과 같으며 여기서, 중요한 부분들을 살펴보면 다음과 같다.

```
main: MODULE

    SEIZE 선언부
    Env: PROCESS( );
       DCL 선언부
       DCL  event_tbl ARRAY( );

       get_tcn_tbl( );
       link_proc ( ); 수행부
       Event 테이블의 offset 계산
       Event 테이블의 특정 시나리오 수행부
       creat_log_file( );
       restore_tcn_tbl( );
    END Env;
    START Env( );
end main;
```

그림 8 main 모듈의 구조

• **SEIZE 선언부** : 별도의 spec 화일이나, 또는 다른 화일에서 선언되거나 정의된 리터럴 중 main 모듈에서 사용되는 리터럴에 대하여 SEIZE해 준다.

• **DCL 선언부** : Env 프로세스에서 사용하는 변수들에 대해서 선언하고 정의한다. 특히 이 중에서 배열 event_tbl은 테스트 슈트를 구성하고 있는 테스트 케이스들의 이벤트들을 테이블 형태로 유지하고 있는 이벤트 테이블이다. 이 이벤트 테이블은 모든 테스트 케이스의 이벤트들을 테스트 케이스의 순서에 따라 차례대로 저장하고 있으며, 이벤트들의 수행은 이벤트 테이블의 offset을 계산하여 특정 테스트 케이스를 지정한 다음 이벤트들을 차례대로 수행하면 된다. 이벤트 테이블의 각 레코드는 [이벤트의 타입, 전송되는 메시지, 소스(또는 타겟) 프로세스, 데이타 초기화 루틴(또는 검사 루틴) 이름] 형태로 구성된다.

• **Link 프로시쥬어 수행부** : 각각의 테스트 케이스마다 수행하기에 앞서 최초 수신 메시지 선언이 필요한 경우가 있는데 선택된 테스트 케이스에 알맞게 메시지들을 link해 주는 일을 수행하는 부분이다.

• **이벤트 테이블의 offset 계산부** : 이벤트 테이블에서 선택된 테스트 케이스의 이벤트들에 대한 위치를 찾기 위한 offset을 계산하는 부분이다.

• **Event 테이블의 특정 시나리오 수행부** : 선택된 테스트 케이스의 이벤트들을 이벤트 테이블의 순서에 따라 차례로 수행시키는 부분이다.

• **Log 화일 생성부** : 선택된 테스트 케이스의 이벤트들의 수행이 끝나면 수행 결과를 기록하기 위하여 log 화일을 작성한다.

Procedure 모듈은 공통적으로 사용되는 함수들을 모아 놓은 모듈이다. 이 모듈의 구조는 다음의 그림과 같다.

```
procedure: MODULE

      SEIZE      선언부
      DCL 선언부

      link_proc 프로시쥬어 선언부
      send_msg 프로시쥬어 선언부
      receive_msg 프로시쥬어 선언부
      timer_proc 프로시쥬어 선언부
      tcn_tbl 관련 프로시쥬어 선언부
      creat_log_file 프로시쥬어 선언부
      creat_acc_log 프로시쥬어 선언부
      check_list 관련 프로시쥬어 선언부

      GRANT 선언부

END procedure;
```

그림 9 procedure 모듈의 구조

Stub 모듈은 입력 데이타 화일로부터 테스트 데이타를 읽어 들여 전송 데이타의 각 필드를 채우기 위한 데

이타 초기화 루틴들과 수신된 데이타의 값들이 기대하던 값들과 일치하는지 혹은 그 범위 내에 있는 지를 검사하는 수신 데이타 검사 루틴들로 이루어진다.

3.4 테스트 드라이버 생성

테스트 스텁은 앞 절에서 나열한 세 개의 모듈이 모두 갖춰지면 컴파일되어 하나의 프로세스(환경 프로세스(Env))로 동작한다. Env 프로세스의 실행 화일과 IUT 프로세스의 실행 화일이 준비되면 드디어 테스팅을 실시할 수 있다. 그림 7의 run-time environment 상에서 Env 프로세스와 IUT 프로세스들의 수행을 지시하는 모듈이 테스트 드라이버이다. 그러나 사실 테스트 드라이버는 run-time environment 위에서 프로세스들의 실행을 지시하는 명령어들의 집합(shell script)이다.

본 연구에서는 run-time environment로 **chillsh**을 사용한다. 테스팅이 준비되면 **chillsh**을 동작시키고 Env 프로세스와 IUT 프로세스 사이의 순서를 고려하여 적절한 순서대로 각 프로세스를 동작시키면 된다. 각 프로세스를 동작시키는 방식은 두 가지가 있다.

첫 번째 방식인 상호동작(interactive) 방식은 Env 프로세스와 IUT 프로세스를 동작시킬 때 사용자가 각 프로세스의 수행을 일일이 지시하는 방식이다. 이 방식은 테스트 데이타를 즉각적으로 변화시켜 가면서 테스팅할 수 있는 특징이 있다. 여기서는 테스트 데이타의 즉각적인 변경이 편리하도록 입출력 창을 가진 윈도우가 사용되어 지며 이를 통해 사용자는 테스트 데이타를 다양하게 입력시킬 수 있다.

두 번째 방식인 배치 방식에서는 사용자는 수행시킬 프로세스들의 순서와 어떤 테스트 케이스를 몇 번 수행시킬 것인가 만을 정해주면 된다(물론 이 과정은 시스템에서 제공하는 사용자 인터페이스를 통해 쉽게 이루어진다). 이렇게 하면 시스템은 배치 명령어 화일을 생성하게 되고 사용자가 이 화일을 **chillsh**에 적재함으로써 프로세스들은 명령어 화일에 기록된 순서대로 **chillsh**에서 차례로 실행된다. 이 방식을 적용하면 테스트 데이타는 입력 데이타 화일에 준비되어 있어야 하고 stub.ch 모듈의 데이타 초기화 루틴 중 배치 방식의 초기화 루틴에 의해 전송 데이타는 초기화된다. 이 방식의 장점은 사용자가 테스팅 과정에 많이 간여하지 않아도 된다는 것이다.

예를 들어, 하나의 Env 프로세스와 세 개의 테스트 케이스(세 개의 시나리오)를 갖는 테스팅 대상 IUT 프로세스 P0, P1이 존재한다고 가정하자. 만일 사용자가 각 시나리오를 1회에 걸쳐서 테스팅해 보려고 한다면 다음과 같은 배치 명령어 화일이 생성된다(여기서 프로세스들의 실행 순서는 P0, P1, Env의 순서이고 사용자가 모든 프로세스들의 수행이 종결될 때마다 10초의 sleeping 시간을 할당했다고 가정하자).

여기서 (run 프로세스_이름)은 지정된 프로세스를 실행시키는 명령이고 (sleep 10)은 chillsh에게 10초 후에 다음 명령어를 수행하라는 명령문이며, (init)은 모두 프로세스가 수행을 종결했을 때 chillsh을 초기화 시켜 주는 명령이다. (quit)는 chillsh의 수행을 종료시키는 명령이다. 이 배치 명령어 화일만 보면 어떤 테스트 케이스를 몇 번 테스팅하는 지 알 수가 없다. 이에 대한 정보는 별도의 테스트 케이스 번호 화일(tcn_tbl)에서 유지된다. Chillsh이 실행되고 이 배치 명령어 화일이 읽혀지면 순서에 의해서 프로세스 P0가 수행을 시작하고 다음으로 프로세스 P1이 수행을 시작하고 마지막으로 Env 프로세스가 수행을 시작한다. Env 프로세스는 테스트 케이스 번호 화일로부터 몇 번째 테스트 케이스를 실행하는 가에 대한 정보를 얻어와서(그림 8의 get_tcn_tbl 부분) 해당 테스트 케이스를 선택(그림 8의 event 테이블의 offset 계산 부분)해서 실행(그림 8의 event 테이블의 특정 시나리오 수행부 부분)한다. Env 프로세스는 수행을 종결할 때 현재 선택되어진 테스트 케이스에 대하여 잔여 수행 횟수를 하나 감소시킨다(그림 8의 restore_tcn_tbl 부분). 이렇게 함으로써 계속해서 이어지는 수행에서 테스트 케이스의 잔여 수행 횟수가 0이면 그 다음 테스트 케이스가 선택되고 선택된 테스트 케이스를 따라서 테스팅 과정은 진행된다.

```
run p0
run p1
run Env
sleep 10
init
run p0
run p1
run Env
sleep 10
init
run p0
run p1
run Env
sleep 10
quit
```

그림 10 테스트 드라이버의 형태

3.5 수행 후 분석

IUT 프로세스들과 테스트 스텁을 한 번씩 수행할 때

마다 발생한 이벤트 시퀀스는 로그 화일에 기록된다. 수행 후 분석(post-mortem analysis) 과정에서는 이렇게 생성된 로그 화일과 원래의 MSC 명세로부터 추출된 이벤트 시퀀스들을 비교하여 구현된 CHILL 프로그램에 오류가 존재하는지를 판단한다.

테스트 스텁이 한 번 수행될 때마다 테스트 스텁은 어느 한 시나리오를 따라 실행된다. 그런데 병렬 프로그램의 특성상 테스트 스텁과 IUT 프로세스들간의 상호작용으로 발생하는 이벤트 시퀀스는 테스트 스텁이 현재 수행하는 시나리오와 일치하지 않을 가능성이 있다. 예를 들어, 세 개의 테스트 케이스를 가진 IUT 프로세스들을 테스팅 한다고 하자. 여기서 테스트 스텁이 첫 번째 테스트 케이스를 따라 실행하고 있을 때 IUT와의 상호작용의 결과는 두 번째 테스트 케이스에 기술된 이벤트 시퀀스를 따를 수 있다. 실제로 각각의 테스트 케이스는 동일한 이벤트 집합으로 이루어져 있고 단지 그 순서만이 다르기 때문에 메시지 도착순서가 바뀌어 첫 번째 테스트 케이스를 수행하려 하여도 두 번째 테스트 케이스를 따르는 결과가 발생할 수 있기 때문이다. 이 상황에서 의도했던 테스트 케이스의 이벤트 실행 순서와 결과로 나온 이벤트 실행 순서가 다르다고 해서 프로그램에 오류가 있다고 판정할 수 없다.

이와 같이 수행 결과가 의도한 테스트 케이스와 일치하지 않을 때 그 결과는 별도로 저장되었다가 테스트 스텁의 모든 수행이 종료된 후에 그 결과들을 원래의 MSC 명세로부터 추출될 수 있는 이벤트 시퀀스들과 비교한다. 그 결과들이 모두 MSC 명세로부터 추출될 수 있는 이벤트 시퀀스들이라면 구현된 CHILL 프로그램의 동작은 명세와 일치한다고 판단할 수 있고 따라서 프로그램 내에 오류가 존재하지 않는다고 결론지을 수 있다. 그러나 일치하지 않은 결과들 중 MSC 명세로부터 추출될 수 없는 이벤트 시퀀스가 존재한다면 구현된 CHILL 프로그램에는 오류가 존재한다고 판단할 수 있다.

4. 구 현

본 연구에서 구현한 테스팅 환경은 UNIX 환경(SUN OS 5.5)에서 C 언어로 작성되었다. 따라서 설치를 위해서 C 컴파일러와 yacc 프로그램이 필요하며, 그래픽 환경이 사용되기 때문에 X/Motif 환경(X11R5/Motif 2.0 이상)이 제공되어야 한다. 본 연구에서는 다수의 도구를 유기적으로 통합하고 사용자와의 편리한 인터페이스를 위해서 그래픽 사용자 인터페이스를 제공한다.

그림 11은 본 테스팅 시스템의 초기 화면과 **MSC 조합 에디터**를 이용해서 MSC를 조합하는 과정을 보여주기 위한 그림이다. 그래픽 환경은 세 개의 윈도우로 구성되어 있다. 상단의 윈도우는 각종 메뉴를 보여주며 환경을 전체적으로 관리하기 위해서 사용되는 메인 윈도우이며 하단 좌측의 윈도우는 그래픽 에디터 사용을 위해서 제공되는 에디터 패널이다. 또한 하단 우측 윈도우는 MSC 조합을 위한 그래픽 에디터이다.

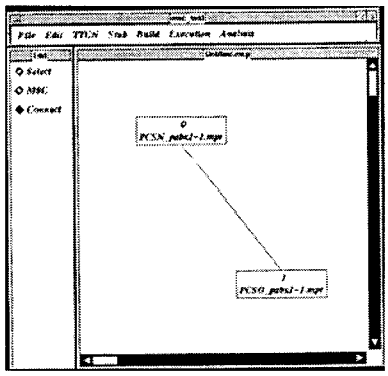

그림 11 테스팅 시스템의 화면 구성

그림 12는 조합된 MSC를 바탕으로 **테스트 케이스를 생성**하기 위해 TTCN 메뉴의 Generate Testsuit 항목을 선택한 경우의 화면을 보여준다. 테스트 케이스를 생성하기 위해서는 MSC 내에 있는 프로세스들을 두개의 그룹으로 나누어야 한다. 즉, 테스팅 대상 프로세스와 그렇지 않은 프로세스를 정해줘야 하는데 그림 12의 대화 상자를 이용해서 프로세스들을 그룹으로 나눈다. 테스팅 하고자 하는 프로세스를 선택하면 그 프로세스의 이름 양쪽에 * 표시가 붙게 된다. 그림의 결과는 line_a, region, trunk의 세 프로세스를 테스팅 대상 프로세스로 선택한 상태이다. 이 상태에서 Apply 버튼을 누르면 선

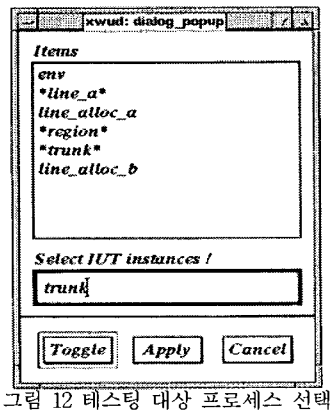

그림 12 테스팅 대상 프로세스 선택

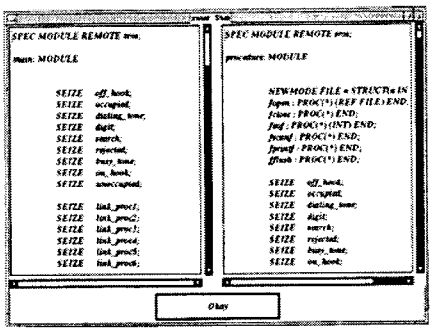

그림 13 생성된 테스트 스텁

택된 프로세스들을 테스팅 하기 위한 TTCN으로 기술된 테스트 케이스가 생성된다.

테스트 케이스 생성 후에 실제 프로그램을 테스팅 하기 위해서는 테스트 스텁과 테스트 드라이버를 만들어야 한다. 그림 13은 주어진 테스트 케이스로부터 **테스트 스텁**을 만들기 위해 Stub 메뉴에 있는 Generate Stub 항목을 선택한 후에 생성된 CHILL 코드 스텁을 보여준다. 그림에서 보는 바와 같이 화면의 왼쪽이 main 모듈이고 오른쪽이 procedure 모듈이다.

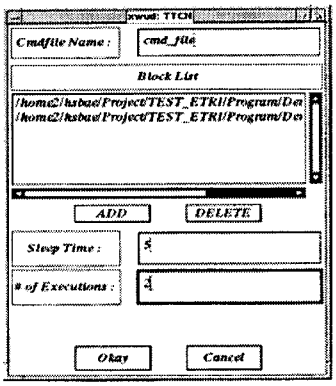

그림 15 테스트 드라이버 생성 정보 입력

그림 14는 **테스트 드라이버를 생성**하기 위해 필요한 정보를 입력하기 위한 대화 상자를 보여준다. 이 대화 상자에는 네 가지 정보를 기입하도록 되어있다. 그림에서 Cmdfile name 항목은 테스트 드라이버 화일의 이름을 의미한다. Block List는 **chillsh** 위에서 수행되어야 할 CHILL 프로그램들을 의미하는데 이는 테스팅 대상 프로그램(IUT)과 테스트 스텁 프로그램을 뜻한다. Sleep Time 이란 반복 수행 시 그들 간의 간격을 의미하는데 여기서 5 라고 준 것은 5초를 기다렸다가 다음 수행을 시작하라는 의미이다. 또한 # of Executions는 각 테스트 케이스를 몇 번씩 수행할 것이지를 결정하는 것인데 이는 병렬 프로그램의 비결정성 때문에 하나의 테스트 케이스를 여러 번 수행해 볼 필요가 있기 때문이다. 여기서는 각 테스트 케이스를 2번씩 수행하도록 하였다. 따라서 테스트 케이스가 일곱 개인 경우에 총 14번의 테스팅이 테스트 드라이버에 의해서 진행된다.

이상과 같은 정보를 입력하고 그 결과로 테스트 드라이버가 생성되면 그래픽 환경에서 즉시 **chillsh**을 구동시켜서 테스팅을 준비해준다. 이때 사용자가 **chillsh**에서 테스트 드라이버를 읽어 들이도록 명령을 입력하면 테스팅이 batch 방식으로 진행된다. 테스트 드라이버에 의해 테스팅 작업이 완료되고 나면 테스팅 과정 중에 테스트 스텁에 의해서 기록된 수행 결과들이 3.5절에서 기술된 방식대로 분석된다.

5. 관련연구

병렬 프로그램 테스팅에 대한 현재까지의 연구는 그 목적에 따라서 크게 다음과 같은 세 방향으로 나뉠 수 있다.

첫째, 비결정성을 가진 프로그램의 재수행성 보장 기법에 대한 연구가 수행되었다[1, 2, 7]. 비결정성을 가진 프로그램은 반복 수행이 어려우므로 테스팅 결과에 대한 신뢰도가 떨어지고 오류의 원인을 찾기가 어려워진다. 그러므로 테스팅이나 디버깅 단계의 핵심기술 중에 하나로 재수행성 보장 기술이 필요하다. 이 연구들은 대부분 프로그램 수행을 통해서 얻은 이벤트 트레이스를 강제 수행시켜서 프로그램의 재수행을 보장하므로 테스팅 기법의 관점에서 볼 때 프로그램 기반 테스팅 기법들로 분류될 수 있다.

둘째, 테스트 커버리지에 대한 연구가 수행되었다[3, 8, 9, 10]. 이 연구들은 도달성 그래프나 TSL[11], CSPE[3] 등과 같이 이벤트들 간의 선후관계가 명시적으로 주어져 있는 경우에 테스팅을 위해서 어떤 이벤트 시퀀스를 선택할 지에 대한 기준을 제시한다. 이 방법들은 테스팅을 위한 별도의 명세 과정이 필요하다는 단점을 내포하고 있다. 또한 병렬 프로그램 테스팅을 위해서는 입력 자료와 이벤트 시퀀스가 필요한데 현재까지는 입력 자료와 이벤트 시퀀스를 함께 고려하는 방법이 제시되지 못하고 있는 실정이다. 따라서 독립적인 테스팅 방법으로 사용되기보다는 다른 테스팅 방법에 연계되어서 사용되는 것이 바람직하다.

마지막으로, 실제 테스트 케이스를 생성하려는 노력이 있다[12, 13, 14]. 이 방법은 수행 시에 얻어진 이벤트

트레이스를 변형시킴으로써 새로운 이벤트 시퀀스를 생성하는 방법[12, 14]과 명세로부터 직접 이벤트 시퀀스를 생성하는 방법[13]으로 나뉘는데 전자의 경우에는 프로그램의 정당성(validity)은 검사할 수 있으나 명세의 실현성(feasibility)을 검사할 수 없다는 문제점이 있다. 또한, 후자의 경우에 명세에 묵시적으로 나타난 이벤트들 간의 선후관계를 파악하는 방법이 없었기 때문에 지금까지는 일부분의 이벤트 시퀀스만을 생성해 왔다.

본 연구에서 개발한 명세 기반 테스팅 환경의 특징은 통신 프로그램에서 반드시 요구되는 명세와 프로그램간의 합치성을 검사하는 기능 테스팅을 수행하기 위하여 명세로부터 직접 이벤트 시퀀스를 생성하고 이를 테스트 케이스로 사용하여 실제 프로그램 수행 중에 이러한 이벤트 시퀀스가 발생하는지를 검사한다는 것이다. 이 방법은 통신소프트웨어 개발 과정에서 산출된 MSC 명세로부터 테스트 케이스를 추출함으로써 위의 연구 [3]이나 [11]에서와 같이 테스팅을 위해 별도의 명세를 작성할 필요가 없다. 본 연구에서 제안한 테스트 케이스 생성 과정은 여러 장의 MSC를 연결하여 완전한 하나의 동작 흐름으로 구성하고 여기에 논리 시간 벡터를 적용하여 이벤트간의 부분 선후관계를 추출함으로써 체계적으로 전체적인 이벤트 시퀀스를 자동 생성할 수 있다는 장점이 있다. 따라서 여기서 생성된 이벤트 시퀀스는 연구 [13]에서 제기된 문제점인 일부분의 이벤트 시퀀스가 아닌 시스템 전체의 이벤트 시퀀스로써 진정한 테스트 케이스의 역할을 수행하게 되어 연구 [3]이나 [11]에서와 같이 다른 테스팅 방법과 연계되어 사용되어야 하는 문제점을 해결함으로써 독자적인 테스팅 방법으로 사용될 수 있다는 것이다. 이 방법은 명세로부터 테스트 케이스를 생성하기 때문에 프로그램 기반 테스팅 방법에서와 같이 생성된 테스트 케이스를 강제로 수행하기보다는 프로그램을 여러 번 수행했을 때 발생하는 시나리오와 생성된 테스트 케이스들간의 합치성을 검사한다. 즉, 테스트 케이스 자체가 테스트 오라클의 역할을 일부분 담당하게 된다. 현재 본 연구에서는 프로그램의 정당성을 검사할 수 있는 방법을 제시하고 있으며, 명세의 실현성을 검사하기 위한 연구가 계획되어 있다.

6. 결 론

본 연구에서는 MSC로 작성된 병렬 프로그램의 요구명세로부터 명세 기반 모듈 테스팅을 위한 테스트 케이스를 자동으로 생성하는 방법을 제안하였고 생성된 테스트 케이스를 이용해서 병렬 프로그램에 대한 테스팅을 수행하기 위한 테스팅 환경을 설계, 구현하였다. 명세로부터 테스트 케이스를 자동으로 생성하기 위해서는 명세 내에 묵시적으로 포함되어 있는 이벤트들과 그들 간의 선후 관계를 파악해야 하는데 이를 위해서 본 연구에서는 논리시간벡터를 MSC 명세에 적용하여 이벤트 간의 선후 관계인 이벤트 시퀀스를 추출하고 이를 테스트 케이스로 사용한다. 생성된 테스트 케이스는 TTCN 형태로 기술되고 이는 다시 CHILL 소스 코드 형태로 변환되어 테스트 대상이 되는 모듈과 상호 동작하면서 테스팅 대상 모듈의 동작이 기술된 요구 명세의 내용과 합치하는 지를 검사한다.

본 연구에서 개발한 테스팅 환경은 병렬 프로그램의 수행을 강제로 조절하는 방법을 쓰지 않는다. 프로그램 기반 테스팅 환경에서는 비결정성을 가진 문장을 제어함으로써 프로그램의 수행 방향을 조절하는 기능을 제공하지만 본 연구의 테스팅 환경은 명세 기반 블랙박스 테스팅 환경이므로 수행 조절 기능보다는 수행 후 분석 방법을 채택하고 있다.

병렬 프로그램의 이벤트 시퀀스에 대한 테스팅은 크게 두 가지 방향에서 접근하는데, 첫째는 "프로그램 수행 중에 발생한 이벤트 트레이스가 정당한가?"이고 다른 하나는 "명세로부터 생성된 이벤트 시퀀스가 실제 프로그램에서 실현 가능한가?"이다. 본 연구에서는 현재 첫 번째 문제에 대한 해결 방법을 제시하고 있으며 두 번째 문제 해결을 위해서는 프로그램 수행 조절 방법이 필요하다. 따라서 향후에 병렬 프로그램 디버깅 기법들에서 제시되었던 수행 조절 방법을 테스팅 환경에 포함하는 방법에 대한 연구가 계획되어 있다.

참 고 문 헌

[1] 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, January 1991.

[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 and R. H. Carver, "A Specification-Based Methodology for Testing Concurrent Programs," *Proc. of European Software Eng. Conf.*, pp. 154-172, 1995.

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

[5] E. Rudolph, J. Grabowski, and P. Graubmann, "Tutorial on Message Sequence Charts (MSC'96)," *Tutorial of the FORTE/PSTV'96 conf.*, October

1996.
[6] ITU-T Recommendation Z.120: Message Sequence Chart (MSC), September 1994.
[7] 배현섭, 김현수, 권용래, 김한경, "검출 후 재수행 기법을 이용한 메시지 기반 병렬 프로그램의 디버깅," 정보과학회 논문지(B), 제23권, 제2호, pp. 146-157, 1996.
[8] E. Itoh, Y. Kawaguchi, Z. Furukawa, and K. Ushijima, "Ordered Sequence Testing Criteria for Concurrent Programs and the Support Tool," *Proc. of Asia Pacific Software Eng. Conf. 1994*, pp. 236-245, Tokyo, 1994.
[9] P. V. Koppol and K. C. Tai, "An Incremental Approach to Structural Testing of Concurrent Software," *Proc. of Int'l Symp. on Software Testing and Analysis*, pp. 14-23, 1996.
[10] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. on Soft. Eng.*, Vol. 8, No. 3, pp. 206-215, March 1992.
[11] D. Rosenblum, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52-61, May 1991.
[12] 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.
[13] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger, "Test Case Specification Based on MSCs and ASN.1," *Proc. of the Seventh SDL Forum 1995*, pp. 307-322, 1995.
[14] 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, Tokyo, 1994.

김 현 수

1988년 서울대학교 계산통계학과(학사). 1991년 한국과학기술원 전산학과(석사). 1995년 한국과학기술원 전산학과(박사). 1995년 ~ 1995년 한국전자통신연구원 박사후연수연구원. 1996년 ~ 현재 금오공과대학교 컴퓨터공학부 조교수. 1999년 ~ 현재 Colorado State Univ. 연구교수. 관심분야는 Software Engineering, Object Oriented Software Engineering, Component-based Software Engineering, Distributed Objects Computing.

배 현 섭

1993년 한국과학기술원 전산학과 학사. 1995년 한국과학기술원 전산학과 석사. 1999년 한국과학기술원 전산학과 박사. 1999년 ~ 현재 한국전자통신연구원 선임연구원. 관심분야는 소프트웨어 테스팅 및 디버깅, 실시간 병렬 프로그램 개발 및 검증, 정형 명세

정 인 상

1987년 서울대학교 컴퓨터공학과 학사. 1989년 한국과학기술원 전산학과 석사. 1993년 한국과학기술원 전산학과 박사. 1993년 9월 ~ 1994년 2월 한국전자통신연구원 박사후연수연구원. 1995년 7월 ~ 1995년 8월 영국 Durham 대학 Center for Software Maintenance 방문연구원. 1994년 3월 1999년 2월 한림대학교 교수. 1997년 8월 ~ 1998년 7월 미국 Purdue대학 SERC 방문교수. 1999년 3월 ~ 현재 한성대학교 정보전산학부 교수. 관심분야는 소프트웨어 테스팅, CBSE, Formal Specification Techniques

권 용 래

1969년 서울대학교 문리대학 이학사. 1971년 서울대학교 대학원 이학석사. 1971년 ~ 1974년 육군사관학교 전임강사. 1978년 미국 피츠버그대학 이학박사. 1978년 ~ 1983년 미국 Computer Science Corporation 연구원. 1983년 ~ 현재 한국과학기술원 전산학과 교수. 관심분야는 실시간 병렬 소프트웨어 검증, 실시간 시스템의 객체지향 기술, 고신뢰도 소프트웨어의 품질 보증

정 영 식

1983년 홍익대학교 전자계산학과 학사. 1993년 한남대학교 전자계산공학과 석사. 1983년 ~ 현재 한국전자통신연구원 선임연구원. 관심분야는 시스템 프로그래밍, 차세대 라우터 기술

이 병 선

1980년 성균관대학교 수학과 졸업. 1982년 동국대학교 전산학과 석사. 1999년 ~ 현재 한국과학기술원 전산학과 박사과정중. 1982년 ~ 현재 한국전자통신연구원 책임연구원. 관심분야는 Software Fault Tolerance, Software Reliability, Object-iriented software testing. Component-based Software engineering

이 동 길

1983년 경북대학교 전자공학과 졸업. 1985년 한국과학기술원 전산학 석사. 1994년 한국과학기술원 전산학 박사. 1985년 ~ 현재 한국전자통신연구원 책임연구원으로 근무. CHILL 컴파일러 및 프로그래밍 환경 개발에 참여하였으며 현재는 객체지향 병행처리 소프트웨이 프로그래밍 환경 개발에 참여하고 있음. 관심분야는 컴파일러 구성론, 프로그래밍 언어론, 실시간 객체지향 병행처리용 소프트웨어 개발 환경