에스테렐 프로그램 디버깅을 위한 테스트 데이터 자동 생성

(An Automated Test Data Generator for Debugging Esterel Programs)

윤 정 한 † 조 민 경 †† 서 선 애 ††† 한 태 숙 ††††
(Jeong-Han Yun) (Minkyung Cho) (Sunae Seo) (Taisook Han)

요 약 에스테렐은 반응형 시스템 설계에 적합하도록 디자인 된 명령형 동기언어이다. 시스템 개발 시에는 디버깅을 위해 다양한 테스트가 필요하다. 반응형 시스템을 테스트 하려면 일련의 입력을 시간의 흐름에 따라 순서대로 나열하여야 한다. 하지만 원하는 목적에 적합한 테스트 데이터를 생성해 주려면 많은 노력이 필요하며, 이 과정에서 오류가 발생하기도 한다. 따라서 디버깅의 특성상 빠르면서 원하는 목적의 테스트 데이터를 쉽게 표현할 수 있는 도구가 필요하다. 본 연구에서는 디버깅에 도움을 줄 수 있는 테스트 데이터 자동 생성기를 개발하였다. 본 연구는 개발자가 원하는 테스트 데이터를 쉽게 표현할 수 있고, 빠르게 테스트 데이터를 만들어내는 것에 초점을 두었다. 또한 사례 연구를 통해 실제 시스템 개발에 우리의 테스트 데이터 생성기를 적용한 예를 보여준다.

키워드 : 에스테렐, 반응형 시스템, 테스트 데이터 생성, 동기 언어, 모델 체킹

Abstract Esterel is an imperative synchronous language that is well-adopted to specify reactive systems. Programmers sometimes want simple validations that can be applied while the system is under development. Since a reactive system reacts to environment changes, a test data is a sequence of input events. Generating proper test data by hand is complex and error-prone. Although several test data generators exist, they are hard to learn and use. Mostly, system designers need test data to reach a specific status of a target program. In this paper, we develop a test data generator to generate test input sequences for debugging Esterel programs. Our tool is focused on easy usage; users can describe test data properties with simple specifications. We show a case study in which the test data generator is used for a practical development process.

Key words : Esterel, reactive system, test data generation, synchronous language, model checking

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2009-C1090-0902-0020) 및 2008년 정부(교육과학기술부)의 재원으로 한국학술진흥재단(KRF-2008-313-D00968)의 지원으로 수행되었음
· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(IITA-2009-C1090-0902-0020)

† 학생회원 : KAIST 전산학과
jeonghan.yun@gmail.com
†† 비 회 원 : LG전자 연구원
chominkyung@lge.com
††† 비 회 원 : 삼성종합기술원 연구원
sunae.seo@samsung.com
†††† 종신회원 : KAIST 전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 2009년 4월 20일
심사완료 : 2009년 8월 25일

정보과학회논문지 : 소프트웨어 및 응용 제36권 제10호(2009.10)

1. 서 론

임베디드 시스템은 시간의 흐름에 따라 변하는 환경에 맞춰 그에 맞는 결과를 계속 내야 하는 반응형 시스템이 많다. 반응형 시스템을 테스트하려고 할 때, 하나의 테스트 데이터를 만들 경우에도 연속적인 외부 환경의 입력들을 원하는 특성에 맞추어 작성하기란 매우 어렵다. 적절한 테스트 데이터는 시스템 개발자가 시스템을 설계/수정할 때 큰 도움을 준다. 하지만, 반응형 시스템을 개발하면서 간단히 사용할 수 있는 테스트 데이터 하나를 만들려고 해도 많은 작업이 필요하다. 기존 테스트 데이터 생성기들, 또는 모델 체커를 이용한 테스트 데이터 생성 기법들을 사용하려고 해도 '원하는 테스트 데이터를 표현'하는 것 자체가 복잡하다.

에스테렐[1-3]을 이용하여 프로그램을 개발/디버깅할

때 개발자가 손쉽게 사용할 수 있도록 하고자 본 연구를 하게 되었다. 여러 가지 시스템을 개발해 본 경험에 따라, 개발자가 원하는 테스트 데이터를 "쉽게" 표현해서 "간단하게" 테스트 데이터를 생성할 수 있는 것이 중요함을 알 수 있었다.

에스테렐에서 모든 동작은 매시간(clock tick)마다의 신호(signal) 상태변화로 나타난다. 이 경우, 시스템 개발자가 원하는 테스트 데이터는

1. 프로그램의 특정 위치에 대한 테스트 데이터
2. 특정 신호들이 동시에 출력되는 테스트 데이터

라고 판단하고, 우리는 테스트 데이터 생성 조건을 "n개의 신호가 동시에 출력되는 상황"으로 간략화 하였다. 프로그램의 특정 위치라는 것도 병렬로 여러 개의 기능들이 수행되는 전체 시스템을 고려하였을 경우, 각 구성요소들마다 원하는 위치에 디버깅을 위한 신호 출력을 삽입하는 것만으로 충분히 간단히 표현할 수 있다.

이 논문에서는 에스테렐 프로그램의 완성도 높은 구현 및 디버깅을 돕기 위해 다른 검증 도구들의 단점을 보완해 줄 수 있는 디버깅 도구를 제안한다. 본 논문에서 제안하는 디버깅 도구는 에스테렐 프로그램을 디버깅하는 방식으로 확인하고자 하는 속성의 상태나 특정 실행 상태에 도달하도록 프로그램 동작을 유도하는 일련의 연속적인 입력들로 이루어진 테스트 데이터를 생성하여 사용자에게 제공하는 방법을 채택하고 있다. 에스테렐 프로그램에서는 입력 신호와 출력 신호의 집합으로 이루어지는 외부 환경과의 상호 작용이 클록(clock) 단위로 동기화되는데, 이러한 에스테렐의 언어적 특성을 고려하여 디버깅 도구를 설계한다.

본 논문은 2장에서 에스테렐에 대해 간략히 소개한 후 테스트 데이터 생성에 관련된 연구들을 살펴본다. 3장에서 본 논문의 테스트 데이터 생성기를 제안하고, 4장에서는 실제 시스템에 적용한 사례를 제시한다. 마지막으로 5장에서 결론 및 향후 연구 방향을 기술한다.

2. 관련 연구

2.1 에스테렐

에스테렐은 동기언어의 한 형태로, 절차 언어(imperative language)의 특징을 갖추고 있는 특징이 있다. 현재 버전 7[3]까지 개발되어 다양한 데이터 타입과 다양한 표현력을 제공한다. 본 논문에서는 에스테렐 프로그램 반응의 기본인 pure signal만을 다루는 에스테렐의 일부분에 대해 이야기 하겠다. 그림 1에서는 본 논문에서 사용하는 에스테렐언어의 문법과 간략한 의미를 보였다.

에스테렐 프로그램의 가장 큰 특징은 pause 문장에서만 시간이 흐른다는 것이다. 나머지 문장들은 실행될 때

`nothing`	no operation
`pause`	time consumption
`emit` S	signal emission
$p;q$	sequence
`present` S `then` p `else` q `end`	signal test
`loop` p `end`	nonterminating loop
$p\|\|q$	parallel execution

그림 1 에스테렐 문법

소모되는 시간은 없다고 가정한다. pause와 pause 사이에 있는 문장들은 모두 동시에 수행되는 것이다.

이와 같은 프로그래밍 모델은 반응형 시스템에서 외부 환경의 입력에 대한 출력 동기화를 표현하기에 적합하다. 실제로 많은 기업에서 에스테렐을 이용하여 시스템을 개발하고 사용하고 있다[4].

2.2 에스테렐 프로그램 테스트 데이터 생성

요구사항에 맞는 반응형 시스템 개발을 위한 검증 및 디버깅 방법에 대한 다양한 연구들이 진행 중이다. 반응형 시스템의 안전성 검증을 위해서는 모델 체킹[5] 기법이 가장 널리 사용된다. 모델 체킹 기법에서 계속 극복해 나가야 할 문제로 다음의 4가지가 있다[6]. 첫째, 실제 시스템을 잘 반영하는 모델 제작. 둘째, 검증하고자 하는 특성의 표현. 셋째, 모델 분석 시 시간/공간에서의 상태 폭발 문제(state explosion problem). 마지막으로 모델 체커 결과물의 해석이다. 이와 같은 어려운 점들을 해결하면서 모델 체킹 기법이 개발 현장에서 사용 가능하도록 많은 연구가 진행되고 있다.

일반적 테스트 데이터 생성에서도 모델 체킹 기법을 사용하고 있다[7-11]. 만들고 싶은 테스트 데이터를 모델 체커가 검증할 특성으로 표현하여 검증 결과로 나오는 반례를 이용하는 것이다. 프로그램의 특정 위치 실행, code coverage 등 다양한 목적에 맞게 모델 체커를 이용하여 테스트 데이터를 생성한다.

에스테렐을 기반으로 한 상용 개발도구인 Esterel Studio[12]에서는 모델 체커를 이용한 테스트 생성 도구를 제공하고 있다. assert와 모델 체커를 이용하여 state coverage, state-pair transition coverage, output coverage, source code coverage, state set coverage 등 다양한 목적을 위한 테스트 케이스를 생성한다. 이러한 테스트 케이스는 시스템 검증의 입장에서 큰 도움이 된다.

모델 체커를 이용한 테스트 데이터 생성[13]은 기존에 있는 자동화 도구들을 이용할 수 있다는 장점이 있다. 하지만 사용상에 어려움이 크다. 같은 특성을 입력해도 사용하는 모델 체커의 내부 알고리즘에 따라 다른 테스트 데이터가 생성될 수도 있다. 그러므로 올바른 사용을 위해서는 모델 체커 내부에 대한 이해가 필요하다. 또한 모델 체커를 사용할 때의 어려움, 즉 검증 목표 작성의 복잡성, 상태 폭발 문제, 모델 체커를 적용할 수 있도록

프로그램 수정이 요구되는 등의 문제점들이 똑같이 발생할 수 있다.

반응형 시스템 개발을 위한 디버깅 도구 연구도 활발히 진행 중이다. 대표적으로 Ludic[14]이라는 디버깅 도구가 있다. Ludic은 브레이크 포인트로 반응형 시스템의 기본 블록 단위를 이용하여 프로그램의 단계적 실행이 소스 코드 단계에서 가능하도록 하였다. 하지만 이 도구는 Lustre[15]라는 언어를 위해 디자인 되었는데, 데이터흐름 동기 언어(dataflow synchronous language)이므로 에스테렐의 절차 언어 특징을 반영하지 못한다.

프로그램 완성 후의 검증 뿐 아니라, 개발 도중에도 시스템에 대한 다양한 정보 제공 및 테스트를 할 수 있는 디버깅 도구도 매우 중요하다. 본 논문에서는 에스테렐의 특징을 잘 반영하면서 개발자가 쉽게 쓸 수 있는 테스트 데이터 생성기를 개발하는데 초점을 맞추었다.

3. 테스트 데이터 자동 생성

3.1 용어 정의

전체 구조 소개에 앞서 전 과정에서 사용할 용어들에 대한 정의가 필요하다.

- Control Flow Graph (CFG): G(N,E), 여기서 N은 a set of nodes, E는 a set of edges
- Node n: 자연수
- Edge e: (Node×Node) → Stmt
- Stmt: NOTHING, PAUSE, EMIT S, PRESENT S_true, PRESENT S_false
- Trace t: a sequence of states
- State s: (clock×input signals×output signals),
  clock: 자연수,
  input signals: δ(signals∪-signals),
  output signals: δ(signals)
- History h: a set of traces
- History Table H: n→h, n:Node, h:History
- Target Trace T: n→h, n:Node, h:History (즉, T(i)는 i번째 loop unroll에서 새롭게 발견한 trace의 집합)

원래의 에스테렐 소스 코드로부터 얻은 CFG는 노드의 집합과 에지(edge)의 집합으로 구성된다. 각각의 노드는 자연수의 식별자로 구별하고, 노드의 타입은 자연수로 정의한다. 에지는 머리 노드와 꼬리 노드의 식별자로 정의하며 프로그램의 변화되는 상태를 의미하는 Stmt를 가진다.

프로그램의 상태(State)는 자연수 타입인 처음부터 해당 상태까지의 클록값(tick), 입력 신호의 집합, 출력 신호의 집합으로 정의된다. 이 때 프로그램의 상태를 구성하는 클록값과 입력 신호 집합, 출력 신호 집합 각각은 Stmt에 의해 변할 수 있다. 일련의 연속적인 프로그램 상태들은 프로그램의 수행 경로(Trace)가 되며, 경로가 모여 기록(History)을 구성한다. CFG내의 각각의 노드는 개별적인 history를 가지며 노드의 식별자와 각자의 history를 대응시켜주는 것이 히스토리 테이블(history table)이다. 루프를 전개할 때마다 히스토리 테이블은 새롭게 갱신되며 그에 따라 찾고자 하는 신호의 발생되는 경로들도 새롭게 산출되는데 이 때 목적 경로(target trace)는 매 루프 전개마다 새롭게 생겨나는 목적 경로의 집합을 저장한다.

3.2 테스트 데이터 생성 도구 구조

그림 2는 본 논문에서 구현한 테스트 데이터 생성 도구의 구조이다. 각 구성요소는 다음과 같다.

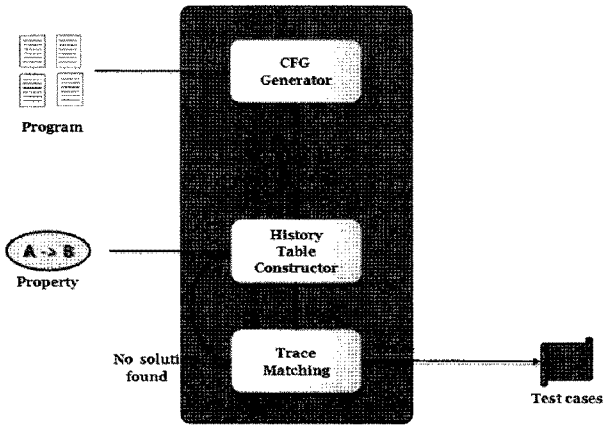


그림 2 테스트 데이터 생성 도구 구조

- CFG generator: 전체 프로그램을 이루는 문장들을 구조적으로 분석하기 위해 소스 코드를 CFG로 변환시켜 CFG상에서 디버깅을 수행한다.
- History Table Constructor: CFG내의 모든 에지를 탐색하여 각 노드마다 시작 노드에서 그 노드까지 도달 가능한 모든 실행 경로들을 모아 테이블의 형태로 저장한다. 또한 이 과정에서 본 도구를 이용해 실행 가능 여부를 확인해 보고자 하는 속성에 맞는 노드가 발견되면 그 노드에까지 도달하기 위한 모든 경로들을 선택하여 따로 추출한다.
- Trace Matching: 이전 과정에서 따로 모아진 목적 경로들에 대해 두 개의 목적 경로가 동시에 만족될 수 있는지 양쪽을 비교한다. 만약 두 개의 경로가 상호 위배되는 과정을 전혀 포함하고 있지 않다면 이 경로들을 합친 것이 얻고자 하는 테스트 데이터가 된다.

프로그램 분석에 유한성을 부여하기 위해 History Table Construction 과정에서 루프가 가지는 후진 에지(back edge)에 대한 탐색은 수행하지 않는다. 다시 말해서 루프의 바디를 이루는 문장들에 대해 단 한 번의 탐색만을 진행하는데, 이 때 원하는 실행 경로가 추출되

지 않거나 추출된 실행 경로들의 대응에 실패하게 되면 다시 한 번 루프의 바디에 대한 탐색을 이전 탐색에 이어서 진행한다. 루프 전개 횟수는 사용자가 선택할 수 있도록 하였다. 또한 소모 클록 수로도 분석 진행의 한계를 표시할 수 있도록 하였다.

3.3 목적 프로그램과 명세

목적 프로그램은 그림 3과 같이 각기 하나의 루프로 이루어진 여러 개의 모듈이 병렬로 연결된 형태로 가정한다. 반응형 시스템의 여러 구성 요소들이 병렬로 동작하는 형태를 반영한 것이다.

문제를 단순하게 하기 위해 루프 바디에는 루프가 없다고 가정한다. 루프가 중첩되어 있는 경우는 그림 4와 같이 목적 프로그램의 형태로 변경이 가능하다.

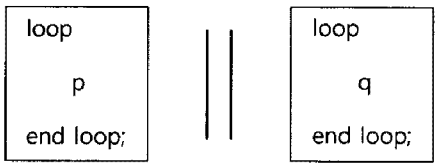


그림 3 목적 프로그램

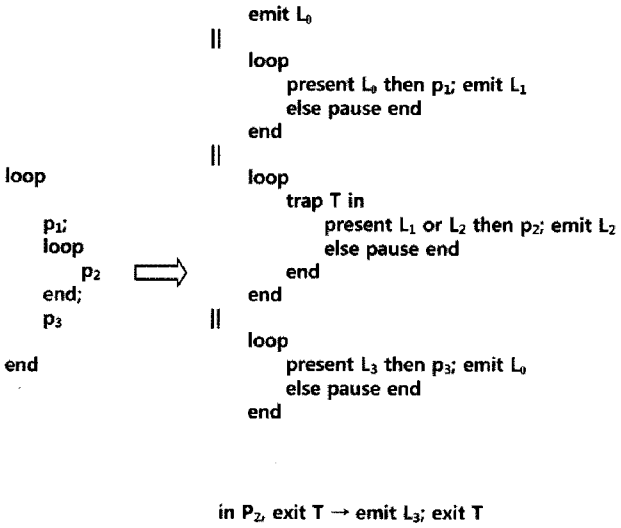


그림 4 중첩된 루프를 목적 프로그램 형태로 변경

3.4 제어 흐름 그래프 탐색

목적 프로그램의 형태에서 병렬로 수행되는 각 모듈을 따로 분석하여 실행 경로를 추출하고, 모듈간의 실행 경로를 비교하여 실제로 수행 가능한 테스트 데이터를 추출하고자 한다. 각각의 모듈마다 제어 흐름 그래프를 만들고, 그래프 상에서 각 모듈별로 실행 경로를 모은다.

한 노드에 최종적으로 모이는 경로 집합은 시작 노드부터 그 노드의 모든 선행 노드들까지 도달 가능한 경로집합들을 모은 것이다. 식별자 m을 가진 노드에서 출발하여 식별자 n을 가진 노드에 도달하는 에지는 상태 stmt:=(m,n)을 가진다. 이 때 stmt에 따라 n의 히스토리가 어떻게 변경되는지 그림 5에서 보여준다. nothing은 아무 것도 하지 않는다. pause는 시간을 1 tick 소모하는 유일한 명령어이다. emit S는 S 시그널이 출력되었음을 알리고, present에서는 입력 시그널의 상태 조건을 추가한다.

3.5 실행 경로 대응

모듈 P와 모듈 Q 각각에 대한 분석을 마치고 나면 신호 Sp와 신호 Sq가 발생하는 상태에 도달할 수 있는 모든 경로들이 양쪽의 목적 경로 집합 내에 모이게 된다. 우리가 찾고자 하는 것은 Sp가 발생하고 나서 n tick 후에 Sq가 발생할 수 있는 실행 경로이다. 그러한 실행 경로를 찾기 위해 Sp를 발생시킬 수 있는 경로와 Sq를 발생시킬 수 있는 경로를 합해야 하며, 이를 위해 양쪽의 집합에 있는 각각의 경로들에 대해 어떤 두 경로가 동시 실행이 가능한 지 서로 대응시켜 보아야 한다. 이 때 대응에 대한 비교 기준 조건은 다음의 세 가지이다.

1. 소모되는 클록 수: Sq를 발생시키는 경로가 소모하는 시간은 Sp를 발생시키는 경로가 소모하는 시간보다 n tick 커야 한다.
2. 입력 신호 집합: 경로를 구성하는 순차적인 상태에 있어 같은 시간(tick)에 정의된 양쪽의 상태에 대해, 한 쪽의 입력 신호 집합에 임의의 입력 신호 A가 있다면 다른 쪽 입력 신호 집합에 -A가 있어서는 안 된다.
3. 출력 신호 집합: 경로를 구성하는 순차적인 상태에 있어 같은 시간(tick)에 정의된 양쪽의 상태에 대해 한 쪽의 입력 신호 집합에 임의의 출력 신호 -A가 있다면 다른 쪽 출력 신호 집합에 A가 있어서는 안 된다.

만약 어떤 두 가지 경로들에 대한 대응 결과가 위의 조건을 위배하지 않는다면 두 경로의 합은 Sp가 발생하

[nothing]	$H'(n)=H(n)$
[pause]	$H'(n)=H(n)\cup\{x \bullet (n+1,\emptyset,\emptyset) \mid x\in H(m)\wedge x=a \bullet (n,I,O)\}$
[emit S]	$H'(n)=H(n)\cup\{a \bullet (n,I,O\cup\{S\}) \mid x\in H(m)\wedge x=a \bullet (n,I,O)\}$
[present S.true]	$H'(n)=H(n)\cup\{a \bullet (n,I\cup\{S\},O) \mid x\in H(m)\wedge x=a \bullet (n,I,O)\}$
[present S.false]	$H'(n)=H(n)\cup\{a \bullet (n,I\cup\{-S\},O) \mid x\in H(m)\wedge x=a \bullet (n,I,O)\}$

그림 5 Stmt에 따른 노드의 경로 변화

고 나서 n tick 후에 Sq가 발생할 수 있는 실행 경로를 의미하는 테스트 데이터가 될 수 있다. 이 테스트 데이터에서 매 tick마다 존재하는 양쪽 경로로부터 합해진 입력 신호들의 집합은 목적 상태로의 도달을 유도하는 외부 환경에 대한 가정이라고 해석할 수 있다. 매 tick마다 대응되는 입력 신호 집합 내의 입력 신호들이 정의된 상태를 가진다고 가정하면 이 테스트 데이터는 원하는 목적 상태를 도출해 내는 실행 경로로서의 역할을 할 수 있다. 상태가 정의되지 않은 입력 신호들은 어떤 상태를 갖던지 목적 상태에 도달하는데 아무런 영향도 미치지 않는다. 사용자는 얻어진 테스트 데이터를 통해 대상 프로그램의 어떠한 부분이, 또는 어떠한 외부 환경에 대한 가정이 프로그램을 목적 상태로 도달시키는지 점검해 볼 수 있다. 만약 검증하고자 했던 목적 상태가 실제 프로그램에서 도달하지 말아야 할 오류 상태라면 사용자는 테스트 데이터의 분석을 통해 오류 상태로 도달하게 하는 프로그램의 버그를 찾아낼 수 있다.

3.6 루프의 전개

모듈 P와 모듈 Q에 대한 CFG를 한 번 탐색한 후에도 목적 경로 대응에 실패하여 원하는 경로를 찾지 못할 수 있다. 또는 적어도 어느 한 쪽 모듈에서 아예 원하는 목적 경로를 찾지 못할 수도 있다. 이 경우 루프 내의 바디를 한 번 더 전개하여 CFG에 대한 분석을 한 번 더 진행하는 방법으로 다시 실행 경로 추출을 시도한다.

이 때 모듈 실행의 시작 상태는 처음 분석 시작 상태와 다르다. 바디를 한 번 수행한 후의 재수행 상태를 정적 분석하는 것이므로 시작 상태는 이전 분석의 마지막 상태와 동일하다. 따라서 루프를 전개하면 시작 노드에서의 경로 집합은 이전 분석의 마지막 노드에서의 경로 집합으로 대체된다.

이 디버깅 도구의 목적은 원하는 상태에 도달 가능한 경로를 추출해 내는 것이므로 목적을 달성할 때까지 루프를 계속해서 전개해 나간다. 그러나 루프 전개를 무한히 실행할 수는 없다. 이를 위해 사용자로부터 결과로 얻어낼 테스트 데이터가 얼마의 시간 안에 실행되어야 하는지 그 수행 시간에 대한 한계를 입력 받는다. 그런 입력이 없다면 적당한 숫자로 루프 전개의 횟수를 제한해야 한다.

정해진 횟수만큼 루프를 전개시켜 분석을 하여도 테스트 데이터를 찾지 못할 수 있다. 이 경우 not found 메시지를 출력한다. 사용자는 제한 범위를 확장하여 테스트 데이터 생성을 다시 시도할 수 있다.

4. 사례 연구 - 철도 건널목 시스템

철도 건널목 시스템은 건널목 위에 기차가 들어오면 차단기를 내리고, 기차가 건널목을 벗어나면 차단기를 올려 주어 사람들이 안전하게 철도 건널목을 지날 수 있게 해 준다. 우리는 에스테렐로 프로그램을 작성하고, 이를 Verilog로 컴파일하여 FPGA에 올려 그림 6과 같이 레고로 만든 철로와 연결하였다.

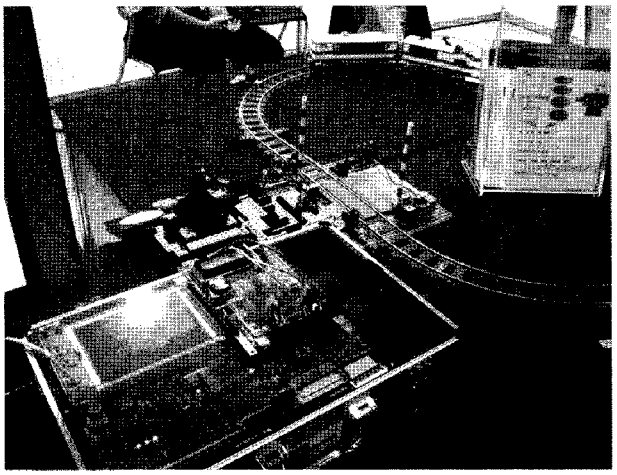


그림 6 철도 건널목 시스템

우리가 목표로 하는 속성은 "기차가 건널목 영역 안에 있으면 항상 차단기를 내린다"이다. 만약 기차가 건널목 안에 있는데 차단기를 내리지 않는 상태를 프로그램에서 찾아낸다면 목적으로 하는 속성을 디버깅하는데 유용할 것이다.

그림 7은 건널목 시스템 프로그램 중 차단기를 컨트롤하는 코드 부분이다. InR 신호는 기차가 건널목 안에 들어와 있는지를 나타내는 것이고, Lower는 차단기를 내리는 명령을 보내는 신호이다. 위에서의 속성을 프로그램 상의 신호로 나타낸다면 "InR과 Lower가 동시에 항상 나타나는가"를 확인하면 된다. 그림 8은 테스트 데이터 생성기를 이용하여 자동 생성한 테스트 데이터들을 보여 준다. 테스트 데이터에서 표시하지 않은 신호는 어떠한 값이어도 관계없는 것을 뜻한다. 이 결과물은 직접 테스트에 사용할 수 있는 형태로도 출력 가능하다. 이 테스트 데이터를 통해 우리는 어떤 경우에 시스템에 오류가 있는지 확인하고, 이를 기반으로 프로그램을 수정할 수 있다.

여기서는 환경에 대한 가정을 고려하지 않아 현실에서는 일어날 수 없는 테스트 데이터도 포함된다. 하지만, 의미상으로는 불가능한 경우도 하드웨어로 컴파일하여 사용할 경우 센서들 간의 타이밍 문제로 인해 일어날 수도 있다. 그러므로 발생 가능한 상황에 대한 충분한 대비가 필요하다. 우리는 이와 같은 정보를 이용해 시뮬레이션에서는 일어나지 않는 오동작이 FPGA상에서 발생할 경우 그 원인을 파악하는데 도움이 되었다.

또한, 철도 건널목 시스템 프로그램에 Esterel Com-

```
module TrainStatus:
  input Enter, Leave;
  output InR;
  relation Enter # Leave;
  loop
    await immediate Enter;
    do sustain InR
    watching immediate Leave;
    emit InR;
    pause
  end
end module

module Controller:
  input HwMode, SwMode, InR, NotClosedState,
        NotOpenedState, SafeRaise, SafeLower;
  output Raise, Lower;
  relation HwMode # SwMode;
  relation NotClosedState # NotOpenedState;
  relation SafeRaise # SafeLower;
  every immediate tick do
    present HwMode then
      present InR and NotClosedState then
        emit Lower
      end;
      present not(InR) and NotOpenedState then
        emit Raise
      end
    end;
    present SwMode then
      present SafeLower and NotClosedState then
        emit Lower
      end;
      present SafeRaise and not(InR) and
              NotOpenedState then
        emit Raise
      end;
      present InR and NotClosedState then
        emit Lower
      end
    end
  end every
end module
```

그림 7 철도 건널목 시스템: 컨트롤러

piler[16]가 instantaneous loop[1]이란 오류를 존재한다고 하였지만, 본 도구를 이용하여 해당 오류가 생기는 테스트 데이터가 없음을 확인하였다. 그 에러는 HwMode 시그널과 SwMode 시그널이 모두 꺼져야 나오는 에러인데, 현실에서 실제 시스템에서는 두 시그널 줄 하나는 꼭 출력되도록 설계하였기 때문이다. 즉, 그 오류 메시지는 시그널 상태 정보를 고려하지 않아 생긴 오류(false alarm)임을 알 수 있었다.

5. 결론 및 향후 연구 과제

프로그램을 개발하면서 현재 수정하거나 새로이 작성한 부분에 대한 테스트는 개발자에게 필수불가결한 요소다. 개발자는 개발 도중에도 특정 조건을 만족하거나 프로그램의 일정 부분을 테스트 할 수 있는 테스트 데이터를 만들어야 한다. 반응형 시스템의 경우, 테스트 데이터는 시간 흐름에 따른 외부 환경으로부터의 입력이 나열되어야 하므로, 하나의 테스트 데이터를 만드는 것도 많은 노력이 필요하다.

본 논문에서는 에스테렐 언어로 작성한 프로그램에서 확인하고자 하는 속성에 도달하도록 프로그램 동작을 유도하는 입력들을 자동으로 생성하는 방법을 제안하였다. 기존 검증기나 모델 체커를 이용한 테스트 데이터 생성기와 달리, 본 논문에서는 개발자가 언제든 빠르고 편리하게 사용할 수 있는 도구에 초점을 맞추었다.

목적 프로그램에 대한 명세를 정의하는데 있어서 다수 모듈들의 병렬적 동작을 가정하였다. 더 정확히는 하나의 loop로 이루어진 두 모듈의 병렬 동작이라는 가정을 두었지만 내부에 또 다른 loop나 병렬화된 문장들이 존재하는 경우도 같은 의미를 가지면서 가정에 맞는 또 다른 프로그램으로 변환하는 방법을 소개하여 해결하였다. 또한 확인하고자 하는 속성에 대한 명세는 사용에 있어서의 편리함을 고려하여 두 개의 신호 이름과 진행 tick수를 의미하는 자연수의 세 개의 인자만으로 간단하

```
(0, {Enter, -Leave, HwMode, -NotClosedState})
(0, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, HwMode, NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, HwMode, NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, HwMode, -NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, HwMode, -NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, SafeLower, -SafeRaise}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, SafeLower, -SafeRaise}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, -SafeLower, SafeRaise, NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, -SafeLower, SafeRaise, NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, -SafeLower, SafeRaise, -NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, NotClosedState, -SafeLower, SafeRaise, -NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, SafeRaise, NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, SafeRaise, NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, SafeRaise, -NotOpenedState}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, SafeRaise, -NotOpenedState}), (1, {Enter, -Leave, SwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, -SafeRaise}), (1, {Enter, -Leave, HwMode, -NotClosedState})
(0, {-Enter, SwMode, -NotClosedState, -SafeRaise}), (1, {Enter, -Leave, SwMode, -NotClosedState})
```

그림 8 건널목 시스템 프로그램을 위해 생성된 테스트 데이터

게 작성할 수 있게 하였다.

향후 연구 목표는 다음과 같다. 먼저, 본 논문에서의 목적 프로그램에 대한 제약을 완화하여 에스테렐의 다양한 특징을 반영시킬 수 있도록 한다. 그리고 명세된 속성에 맞는 목적 상태에 도달하기 위한 과정에 있어 시간적, 공간적 복잡도를 낮출 수 있는 요약 기법을 적용해 보고자 한다. 이는 프로그램의 디버깅 및 이해를 돕는데도 활용될 수 있을 것이다.

참 고 문 헌

[1] G. Berry, *The Constructive Semantics of Pure Esterel,* Draft book available at http://www.inria.fr/meije/esterel/esterel-eng.html, 1999.

[2] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*, Springer, 2007.

[3] Esterel Technologies, *The Esterel v7 Reference Manual Version v7.30. initial IEEE standardization proposal*, Esterel Technologies, 679 av. Dr. J. Lefebvre 06270 VilleneuveLoubet, France, November 2005.

[4] Esterel Technologies. Success stories, http://www.esterel-technologies.com/technology/success-stories/

[5] E. M. Clarke, Jr. O. Grmberg, and D. A. Peled, *Model Checking*, MIT Press, Cambridge, MA, 2000.

[6] J. Hatcliff and M. Dwyer, using the Bandera Tool Set to Model-check Properties of Concurrent Java Software, *Proceedings of CONCUR 2001 (LNCS 2154)*, 2001.

[7] A. Gargantini and C. Heitmeyer, Using Model Checking to Generate Tests from Requirements Specifications, *Software Engineering Notes*, 24(6) pages 146-162, November 1999.

[8] P. E. Ammann and P. E. Black, A Specification-Based Coverage Metric to Evaluate Test Sets, In *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, November 1999.

[9] H. Hong, S. Cha, I. Lee, O. Sokolsky, and H. Ural, Data Flow Testing as Model Checking, In *Proceedings of the International Conference on Software Engineering*, May 2003.

[10] H. Hong, I. Lee, O. Sokolsky, and H. Ural, A Temporal Logic Based theory of Test coverage and Generation, In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, April 2002.

[11] S. Rayadurgam and M. P. E. Heimdahl, Coverage Based Test-case Generation Using Model Checkers, In *Proccedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp. 83-91, April 2001.

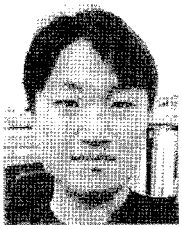
[12] Esterel Studio, http://www.esterel-technologies.com/support.php?rub=86

[13] G. Devaraj, M. P. E. Heimdahl, and D. Liang, Coverage-Directed Test Generation with Model Checkers: Challenge and Opportunities, *The 29th Annual International Computer Software and Applications Conference*, 2005.

[14] F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi, Automatic state reaching for debugging reactive programs, In *Proceedings of 5th International Workshop on Automated Debugging*, September 2003.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language Lustre," *Proceedings of the IEEE*, vol 79(9), pp.1305-1320, 1991.

[15] The Esterel v5.92 Compiler, http://www-sop.inria.fr/esterel.org/



윤 정 한

2001년 KAIST 전산학전공 학사. 2003년 KAIST 전산학전공 석사. 2003년~현재 KAIST 전산학과 박사과정. 관심분야는 임베디드 시스템, 프로그램 분석, 프로그래밍 언어

조 민 경

2007년 서강대학교 컴퓨터학전공 학사 2009년 KAIST 전산학전공 석사. 2009년~현재 LG전자 연구원. 관심분야는 임베디드 시스템, 프로그래밍 언어

서 선 애

1998년 KAIST 전산학전공 학사. 2000년 KAIST 전산학전공 석사. 2007년 KAIST 전산학전공 박사. 2009년 KAIST 전산학전공 박사후 연구원. 2009년~현재 삼성종합기술원 연구원. 관심분야는 프로그래밍 언어, 프로그램 분석, 임베디드 시스템 분석

한 태 숙

1976년 서울대학교 전자공학과 학사 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사. 1991년~현재 KAIST 전산학전공 교수. 관심분야는 프로그래밍 언어, 함수형 언어, 임베디드 시스템