임베디드 시스템의 자동 테스팅을 위한 테스트 수행기 스케줄링

정성욱 [O], 최경희*, 정기현**
[O] 아주대학교 정보통신전문대학원
*아주대학교 정보통신전문대학원
**아주대학교 전자공학부

정성욱 dakkogi@rpa.re.kr
최경희 khchoi@ajou.ac.kr
정기현 khjung@ajou.ac.kr

Scheduling of Test Executor For Automatic Embedded System Testing

Sung-Wook Jung[O], Kyung-Hee Choi*, Ki-Hyun Jung**
[O]Graduate School of Information and Communication, Ajou University
*Graduate School of Information and Communication, Ajou University
**Division of Electronics Enginnering, Ajou Unniversity

요 약

본 논문에서는 요구사항 기반 신뢰성 자동 테스트를 하기 위해서 테스트 수행기를 구현하고. 테스트 수행기 내부의 스케쥴링 방법을 제안하였다. 제안한 방법으로 상용 FATC 를 테스트한 결과 이산적으로 모델링할 수 있는 임베디드 시스템에서는 잘 동작함을 확인할 수 있었다.

1. 서 론

임베디드 시스템의 응용 범위가 확대됨에 따라 임베디드 소프트웨어의 신뢰성을 확보하기 위한 다양한 연구가 이루어지고 있다. 요구사항 기반 소프트웨어 테스팅 (requirement based software testing)[1]은 소프트웨어의 요구사항을 근거로 소프트웨어의 정확성과 신뢰성을 테스팅하는 방법으로서 가장 널리 사용되는 방법의 하나이다.

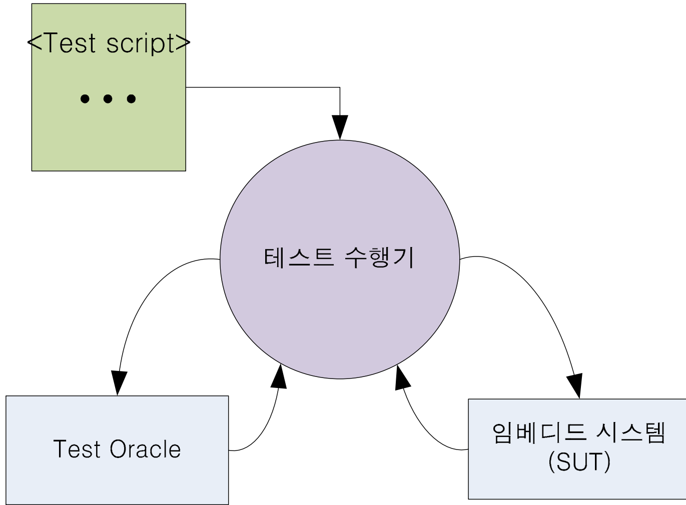
임베디드 시스템의 기능이 복잡해 짐에 따라 소프트웨어의 개발 비용이나 유지보수 비용이 크게 증가하고 있다. 뿐만 아니라 복잡한 기능을 구현하는 소프트웨어의 테스트에 매우 많은 비용이 든다. 더욱이 시스템의 신뢰성 테스트는 많은 부분 사람이 직접 수행하는 경우, 시간과 비용면에서 비효율적이다[2]. 이러한 이유로 신뢰성 자동 테스트가 매우 필요하다.

신뢰성 자동 테스트를 하기 위해서 본 논문에서는 테스트 수행기를 구현하였다. (그림 1)과 같이 테스트 수행기는 요구 사항을 해독하기 위해 변환한 테스트 스크립트를 입력으로 받고, 테스트 스크립트를 오라클과 실제 임베디드 장치에 주입하여 결과를 비교하기 때문에, 테스트 수행기 내부의 스케쥴러의 동작에 의해 결과가 달라질 수도 있다. 따라서 테스트 수행기 내부의 스케쥴러를 잘 구현하는 것은 중요하다.

본 논문에서는 요구사항 기반으로 임베디드 소프트웨어의 신뢰성을 자동으로 테스트하기 위한 테스트 수행기에서의 스케줄링 방법에 대하여 제안한다. 2 장에서 테스트 수행기의 구조를 설명하고, 3 장에서는 테스트 수행기에 주입하는 테스트 스크립트의 문법과, 여러 가지 테스트 스크립트를 제안한다. 4 장에서는 테스트 수행기의 스케쥴링 방법을 설명한다. 마지막으로 구현한 테스트 수행기를 실제 임베디드 시스템에 적용한 사례를 소개한다.

2. 테스트 수행기 구조

테스트를 수행하기 위해서 테스트의 대상이 되는 임베디드 시스템, 즉 System under test(SUT)가 있어야 하며, SUT 의 동작을 검증할 테스트 오라클이 있어야 한다[3,4]. 또한, SUT 에 테스트 과정을 정의한 테스트 스크립트가 필요하다. SUT 와 Test Oracle 과 테스트 스크립트를 이어주는 역할을 하는 것이 테스트 수행기 이다.



(그림 1) 테스트 수행기와 관련 구조

(그림 1)에서와 같이, 테스트 수행기는 테스트 스크립트를 해독한다. 해독된 테스트 스크립트를 한 줄씩 수행한다. 테스트 수행기는 해독된 테스트 스크립트내에 포함된 다양한 명령을 순서대로 수행한다.

스크립트는 SUT 와 테스트 오라클의 특정한 값을 주입하는 것을 명시할 수도 있으며, SUT 와 테스트 오라클의 값을 비교하도록 명시할 수 도 있다. 테스트 스크립트는 간단한 문법적 규칙을 가지는 스크립트이며, 테스터에 의해 수동으로, 혹은 요구사항으로부터 자동으로 생성될 수도 있다. 테스트 스크립트의 문법과 *operation* 은 아래 절에서 설명하기로 한다. SUT 는 대상 임베디드 시스템이다. 임베디드 시스템에 특정한 값을 특정한 시점에 주입하고, SUT 의 값을 읽어올 수 있도록 하기 위해서, SUT 와 테스트 수행기 사이에 별도의 임베디드 장치를 설치했다. 테스트 오라클은 대상 임베디드 시스템의 행동을 정의한 것을 의미한다.

3. 테스트 스크립트

SUT 와 테스트 오라클을 비교하기 위해, 테스트 수행기에 주입하는 스크립트가 테스트 스크립트다. 테스트 스크립트는 SUT 의 입/출력을 변수화한 object 와, SUT 에 대한 직접적인 동작을 기술하는 *operation*, obejct 에 할당하는 value 가 하나의 동작을 기술한다.

(그림 2) 테스트 스크립트 명령어 형식

SUT 에 다양한 입/출력에 대해 테스터가 충분히 다양한 값을 주입하기 위해서는 *object* 와 *operation* 이 충분히 SUT 의 동작을 기술할 수 있어야 한다. *Object* 는 요구사항 모델링에 따라 달라지며, *operation* 은 *object* 의 성격에 의해 결정된다.

<표 1> object 가 반드시 필요한 opeation 들

Operation	사용방법	대상
Write()	Object.*Write*(value)	SUT, Test Oracle
Press()	Object.*Press*()	SUT, Test Oracle
LPress()	Object.*LPress*(value)	SUT, Test Oracle

<표 2> object 를 가지지 않는 opration 들

Operation	사용방법	대상
Wait()	*Wait*(value)	SUT, Test Oracle
Check()	*Check*()	SUT, Test Oracle

<표 1>과 <표 2>는 테스트 스크립트에서 사용할 수 있는 *operation* 의 예를 나타낸다. <표 1>은 *object* 가 반드시 필요한 *operation* 들이다. Write()는 *object* 에 value 를 쓰는 *operation* 이다. SUT 와 테스트 오라클 양쪽에 *value* 의 값을 주입하는 역할을 한다. Press()는 버튼을 누르는 동작을 기술하는 *operation* 이다. 버튼은 상승 에지에서 trigger 가 발생할 수 있고, 하강 에지에서 trigger 가 발생 할 수 있는데, 이는 버튼의 성격에 따라 다르다. 버튼을 모델링한 *object* 를 정의할 때, 상승 에지(0->1)에서 trigger 가 발생하는지, 하강 에지(1->0)에서 trigger 가 발생하는지 기술하여야 한다. Press()는 테스트 수행기 내부적으로 Write()로 치환한다. 상승 에지에서 동작하는 버튼의 Press()일 경우 Write(0) *opreation* 을 SUT 와 테스트 오라클에 주고, SUT 까지의 전파지연시간을 감안하여 쉰 다음에 다시 Write(1) *operation* 을 SUT 와 테스트 오라클에 준다. LPress()는 길게 누르는 버튼에 해당하는 *opreation* 이다. 길게 누르는 버튼의 경우, Press()와 달리, 버튼을 모델링한 *object* 에 일정한 값을 주입시키고, 값을 변화시키지 않으면 된다. 따라서 버튼을 특정시간 동안 눌러놓는 *operation* 의 경우 LPress(1)로 스크립트에 기술하고, 버튼에서 손을 뗄 때, LPress(0)으로 스크립트에 기술한다. <표 3>은 Press()와 LPress()가 테스트 수행기에서 어떻게 인식하는지 보여준다.

<표 3> Press(), lPress()의 동작을 테스트 수행기에서 인식할 때 변환되는 operation

Script 에 기술할 때	테스트 수행기 내부 인식
A.Press() (A 가 상승에지에서 동작하는 버튼일 때)	A.Write(0) 지연시간 A.Write(1)
B.Press() (B 가 하강에지에서 동작하는 버튼일 때)	A.Write(1) 지연시간 A.Write(0)
C.lPress(1) (C 를 계속 누르고 있을 때)	C.Write(1)
C.lPress(0) (C 를 그만 누를 때)	C.Write(0)

Wait()는 SUT 나 테스트 오라클에 어떤 값을 주입하기 위한 *operation* 이 아니다. Wait()는 SUT 에 주입한 값이

충분히 반영될 수 있게 하기 위한 시간만큼 SUT 와 테스트 오라클을 지연시키기 위한 *operation* 이다. 예를 들어 요구사항에, <버튼 A 를 누르고 10 초 후, 시스템의 상태가 S1 에서 S2 로 바뀐다>라고 기술되어 있을 경우, 테스트 스크립트는 A.Press()를 먼저 기술하고, Wait(10)을 기술하여, 버튼 A 가 눌려진 후 10 초 뒤의 시스템 상태를 관찰해야 한다.

Check()는 SUT 와 테스트 오라클의 값을 비교하기 위한 *operation* 이다. 요구사항을 테스트 스크립트로 바꾸어 SUT 와 테스트 오라클을 동작 시킨 후, SUT 가 요구사항대로 동작하는지 알아보기 위해 필요하다. SUT 와 요구사항의 결과가 다르면, 테스트 오라클이 제대로 요구사항을 반영하는지 알아보고, 테스트 오라클에 문제가 없다면, SUT 가 요구사항대로 동작하지 않는다고 결론 내릴 수 있다.

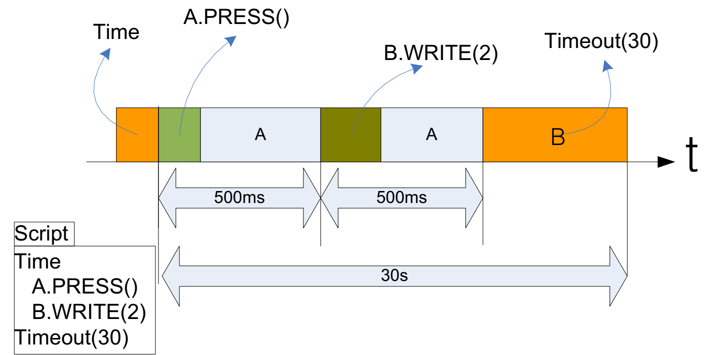
<표 1>과 <표 2>에서 정의한 *operation* 외에도 요구사항을 더욱 정확하게 표현하기 위한 *operation* 이 필요하다. 먼저, Par/Parend 는 SUT 에 값을 순차적으로 주입하는 것이 아니라, 의미적으로 동시에 값을 입력해야 하는 경우에 사용한다. 위에서 설명한 명령행들을 이용하여 테스트 스크립트를 작성하는 경우, 테스트 스크립트는 한 줄씩 순차적으로 실행된다.

<표 4>그 외의 operation

Operation	의 미	대상
Par/Parend	Par 가 나오면 Parend 가 나올때까지 모든 operation 은 동시 입력으로 처리한다.	테스트 수행기
Time/Timeout(value)	Time 과 Timeout(value)사이의 시간적 간격이 Value 이상 됨을 보장한다.	테스트 수행기

그러나, 만약 SUT 나 테스트 오라클로, 테스트 수행기가 여러 동작들을 의미적으로 동시에 수행해야 되는 경우에는, 위에 기술한 *operation* 들로써는 표현이 어렵다. 그래서 Par/Parend 가 필요하다. Par *operation* 을 먼저 테스트 스크립트에 기술하고, Parend 를 테스트 스크립트에 기술할 때까지, Par 와 Parend 사이의 모든 명령행은 동시 입력으로 테스트 수행기에서 처리한다.

또 Time/ Timeout(value)가 있다. 이 *operation* 은 Time 명령어와 Timeout(value)명령어 사이의 시간적 간격이 Value 이상 됨을 보장한다. 즉 Time 을 기술하고 <표 1>과 <표 2>의 명령행들을 기술한 다음, Timeout(Value)를 기술하였다면, Time 부터 Timeout(Value)까지의 시간이 Value 이상이 됨을 의미한다. Time/Timeout(value)는 일련의 명령행들을 처리한 시점에서 특정 시간 후에 SUT 의 결과를 보고 싶을 때, 주로 사용한다. 예를 들어, (그림 3)의 왼쪽 아래와 같은 스크립트 명령에 대하여 생각해보자. 테스트 수행기에서 A.PRESS()를 처리하는데 500ms, B.WRITE(2)를 처리하는데 500ms 가 소요되었기 때문에, timeout(30)에서는 29 초를 WAIT 하게 된다.



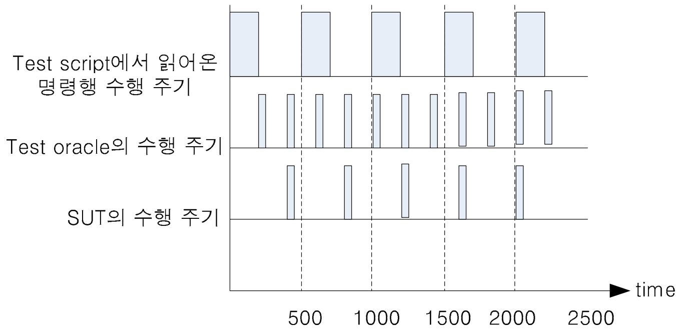
(그림 3) Time/Timeout 의 예제

즉, Time/Timeout(value)는 Time 과 Timeout(value)가 설정된 구간에 속하는 명령행들의 수행시간이 최소한 value 이상이 되는 것을 보장한다. Time 과 Timeout(value)사이에 수행되는 명령행들이 value 시간보다 적은 시간에 수행 종료되면, value 시간에서 수행 종료된 시간을 뺀 시간 만큼 Wait 한다. 이 경우가 (그림 3)의 경우이다. 그러나 만약, 수행되는 명령행들의 완료 시점이 value 시간을 초과하는 경우에는 스케쥴러의 현재 수행할 명령어인 Timeout(value)를 스케쥴러에서 삭제한다. Timeout(value)는 일종의 WAIT(value)과 비슷한 동작을 하기 때문에, WAIT 할 시간이 없다면, 무시한다. WAIT(value)는 반드시 value 시간만큼 시간을 보내는 것이지만, timeout(value)는 이전 명령행의 수행시간을 고려해서 WAIT 를 수행하는 것이다..

4. 테스트 수행기의 스케쥴링

SUT 의 행동의 정확성을 판단하기 위하여는 테스트 입력을 주입한 후, SUT 의 행동을 측정하고, 이를 테스트 오라클과 비교분석하여야 한다. 이 때, 테스트 수행기는 대상인 SUT 에 연속적인 값을 주입하기 보다는 이산적인 값을 주기적으로 주입하게 된다. 특히 다양한 환경에서의 임베디드 시스템의 정확성을 판단하기 위하여, SUT 와 테스트 오라클의 상태가 변화될 수 있도록 테스트 수행기는 주기적으로 값을 주입하여 임베디드 시스템을 적절한 상태가 될 수 있도록 SUT 와 테스트 오라클의 상태를 변화시켜야 한다.

테스트 수행기는 테스트 스크립트의 명령 행을 한 줄 읽어와서 실행 시키자 마자 다음 명령 행을 읽어 오지 않는다. 테스트 스크립트에서 <표 1>이나 <표 2>에 속하는 *operation* 을 수행할 때는, SUT 에 값이 충분히 전달되고, 테스트 수행기까지 응답시간을 고려해야 하기 때문이다. 테스트 스크립트의 명령행일 수행하는 경우, 대략 500ms 정도의 시간적인 여유가 있으면, 테스트 수행기가 동작하는데 충분하였다.



(그림 4) 테스트 수행기의 스케쥴링

본 논문에서는 시분할 스케줄링을 사용하여 테스트 실행기의 주요 작업을 스케줄하는 방안을 제안한다. 예를 들면, 테스트 수행기가 테스트 스크립트의 한 명령 행과 다음 명령 행 사이의 실행 간격은 500ms 로 정하여 SUT 를 테스트 하였다. 테스트 오라클은 하드웨어 장치가 아니기 때문에, 500ms 보다 작은 200ms 로 정하였으며, SUT 에서 값을 수집하는 주기는 400ms 로 하였다. (그림 4)는 테스트 수행기의 스케쥴링 모습을 나타낸다.

테스트 스크립트에서 읽어온 명령 행 수행 주기는 테스트 수행기가 Press(),Write()등과 같은 명령어를 처리할 때, SUT 와 테스트 오라클로 값을 주입하고, 응답을 받는 시간을 사각형 박스로 나타내었다. 즉, 테스트 스크립트의 명령행 하나를 수행하는데 500ms 내에서 처리가 끝남을 의미한다. 만약 500ms 내에서 처리가 끝나지 않을 경우 명령행과 다음 명령행의 수행 간격을 500ms 보다 조금 더 크게 설정해서 시뮬레이션 해야한다. 테스트 오라클과 SUT 의 수행 주기는, 테스트 오라클과 SUT 에 값을 주입한 후(명령행의 처리에 의해), 테스트 오라클과 SUT 의 상태를 변화시켜 주는 주기 이다.

스케쥴러에서 고려해야하는 변수는 세 개 이다. 1) 테스트 스크립트에서 읽어오는 명령행 수행주기(ti) 2)테스트 오라클의 수행주기(to) 3) SUT 의 수행주기(ts)가 그것이다. 세가지 변수에 대해 각각 초기값은 ti=0, to=테스트 오라클 수행주기, ts=SUT 수행주기이다. 스케쥴러는 자신이 가지고 있는 세가지 변수 중 가장 작은 숫자가 할당된 변수의 동작을 수행하면 된다. 수행 후, 해당 변수는 해당 변수의 수행 주기를 더해서 업데이트 한다.

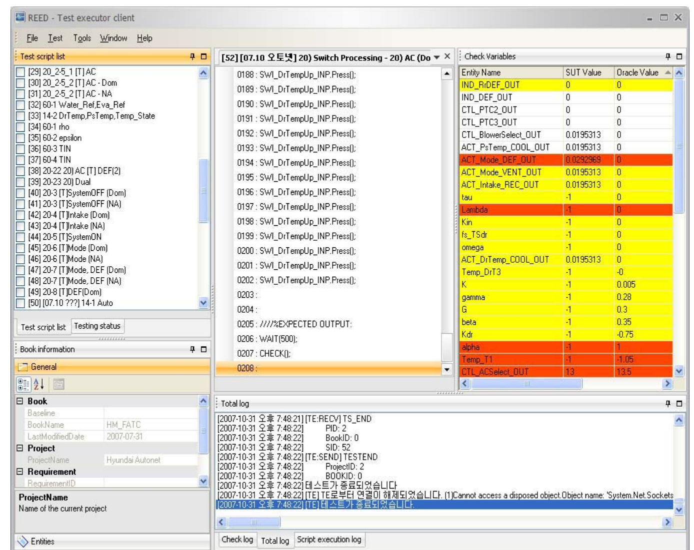
본 논문에서 제안한 스케쥴러는 SUT 의 특성에 유연하게 대처할 수 있는 장점이 있다. SUT 에서의 응답시간이 길어지는 SUT 의 경우에는 테스트 스크립트에서 읽어오는 명령행 주기를 보다 길게 정하면 되고, 반대의 경우에는 짧게 정하면 된다. 테스트 스크립트에서 하나의 명령행과 다음 명령행 사이에는 ti 만큼 시간 간격이 항상 존재하기 때문에, ti 를 적절하게 조정함으로써, SUT 의 변화를 충분히 반영할 수 있다.

또한, ti 동안 테스트 오라클의 상태를 변화시키는 횟수와, SUT 의 상태를 변화시켜주는 횟수를 보장할 수 있다. 예를 들어 (그림 4)에서 500ms 이전의 상태를 보면, 테스트 스크립트에서 명령행을 하나 읽어와서 테스트 수행기에서 처리한다음, 다음 명령행을 읽어올때까지, 테스트 오라클의 상태를 변화시키는 동작을 두번하고, SUT 의 상태를 변화시키는 동작을 한번한다. 만약, 테스트 스크립트에서 명령행을 읽어와서 처리한 다음, 테스트 오라클의 상태를 변화시키는 동작을 처리하지 않는다면, 잘못된 테스트 결과를 도출하게 될 것이다.

5. 적용사례

제안한 테스트 스크립트와 스케줄러를 구현한 테스트 수행기를 상용 자동차에 들어가는 full-automated temperature controller(FATC)에 적용하였다. FATC 를 테스트하기 위해, FATC 의 전체 요구사항을 3 장에서 기술한 테스트 스크립트로 표현하여, 테스트 수행기의 입력으로 주입하였다. 각 테스트 스크립트를 테스트 수행기에 주입하고 구동시켰으며, 그 결과를 (그림 5)의 툴로 확인하였다. (그림 5)의 툴은 테스트 수행기가 테스트 스크립트에서 CHECK *operation* 을 읽어서 수행할 때 마다 테스트 오라클과 SUT 로부터 출력변수들의 값을 보여준다.

총 1682 개의 테스트 스크립트를 테스트 수행기가 수행하였으며, CHECK *operation* 은 총 4319 번 호출되었다. 4319 개의 CHECK 결과 중 1169 개(27%)는 테스트 오라클과 SUT 의 결과가 일치하였고, 3143 개는 테스트 오라클과 SUT 의 결과가 일치하지 않았다. 일치하지 않은 3143 개를 분석해본 결과, 대부분 SUT 의 오류이거나, 요구사항의 잘못된 기술에서 비롯한 테스트 오라클의 오류로 판명되었다.



(그림 5) CHECK operation 시 테스트 오라클과 SUT 의 값 비교

4 장의 마지막 부분에, 제안한 스케쥴링 방법의 장점을 기술하였다. 만약, 스케쥴러가 잘못 설계되어 동작하게 된다면, 테스트 수행기는, SUT 로부터 제대로 응답받기 전에, 다음 명령어를 수행하여, 테스트 결과가 잘못 나오거나, 테스트 수행기에서 명령어 하나를 처리하고 나서, 테스트 오라클과 SUT 의 상태를 변이시키는 동작

을 제대로 하지 않아서 테스트 결과가 잘못 나올 수 있다. 그러나 테스트 결과를 분석해본 결과, 스케쥴러의 잘못된 설계에 기인한 오류는 하나도 없었다.

6. 결 론

본 논문에서는 임베디드 시스템의 자동 테스트를 위한 테스트 스크립트의 문법을 정의하였고, 테스트 스트립트의 *operation* 을 정의하였다. 정의한 테스트 스크립트의 문법과 *operation* 을 이용하여 테스트 수행기의 스케줄링 방법을 제안하였다. 스케쥴러는 SUT 의 응답시간을 충분히 고려하여 설계되었기 때문에, SUT 의 변화와 SUT 로부터 출력되는 값을 테스트 수행기에서 충분히 반영할 수 있는 장점이 있으며, 테스트 오라클과 SUT 의 상태를 변화시키는 동작의 횟수를 보장함으로써, 보다 테스트를 정확하게 수행할 수 있다.

제안한 방법을 실제 임베디드 시스템(FATC)에 적용하여 요구사항과 SUT 의 동작을 비교한 결과, 요구사항과 SUT 가 다르게 동작하는 부분을 찾을 수 있었으며, 이는 테스트 수행기의 구현 목적에 부합하는 결과로 여겨진다. 실험 결과, 약 3000 개의 오류 중에 테스트 수행기의 스케줄러에 기인한 오류는 하나도 없을 정도로 제안한 테스트 수행기의 스케줄링은 잘 동작함을 확인 할 수 있었다.

그러나 제안한 방법은 임베디드 시스템의 테스트를 이산적으로 수행하는 방식이기 때문에, 임베디드 시스템이 이산적으로 모델링되지 않을 경우에는 적용하기 어려운 단점이 있다.

7. 향후 과제

테스트 수행기를 임베디드 장치에 적용하여 성공적으로 테스트를 수행하였다. 그러나 연속적으로 모델링해야 하는 임베디드 시스템의 경우에도 스케쥴링할 수 있도록 알고리즘의 보완이 필요하다. 또, 테스트 수행기가 해석할 수 있는 테스트 스크립트의 *operation* 이 제한적이며, 대상 임베디드 시스템의 동작을 충분히 시뮬레이션 하지 못하는 경우가 생길 수 있다.

참고문헌

[1] Mogyorodi, G., "Requirements-based testing: an overview," Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on , vol., no., pp.286-295, 2001

[2] Kelvin Rodd, " Practical Guide to Software System Testing" , K.J. Ross & Associates Pty. Ltd, 1998.

[3] D. Hoffman, " A Taxonomy for Test Oracle" http://www.softwarequalitymethods.com/Papers/OracleTax.pdf.

[4] Howden, William E. " A Functional Approach to Program Testing and Analysis." IEEE Transactions on software Engineering 12(1986):997-1005