Data-Driven 방식의 효과적인 임베디드 S/W 테스트 방법에 관한 연구

권규환
LG CNS

The Effective Test for Embedded S/W by using Data-Driven Method

Kwon, kyu hwan
LG CNS Company
E-mail : kyukwon@lgcns.com

요 약

전자, 자동차 등 엔지니어링 컨버전스 산업이 발전함에 따라 임베디드 S/W 테스트의 중요성이 증가하고 있다. 그러나, 일반적인 S/W 테스트 방법을 그대로 이용할 경우 임베디드 디바이스의 특성으로 인해 일반적인 품질 수준의 테스트 결과를 얻기 위해 상대적으로 더 많은 비용과 시간을 필요로 하게 된다. 따라서, 다양한 임베디드 시스템의 환경에 적용하기 쉽고, 임베디드 디바이스의 특성에 잘 대응하는 테스트 방법이 요구되는 실정이다. 본 논문에서는 Data-Driven 기법을 이용한 효과적인 임베디드 테스트 자동화 기법을 제안한다.

1. 서론

임베디드 시스템의 특징은 H/W의 제한된 자원과 S/W의 실시간 동작에 있다. 임베디드 S/W 개발이 일반적인 S/W 개발과 구분되는 특징은 개발환경과 운용환경이 다르다는 것이다. 이런 특성은 테스트 방법도 구분 짓는데, 기존의 테스트 방법과 달리 개발환경과 운용환경이 나뉜 임베디드 S/W는 개발환경에서 수행한 테스트를 운용환경에서 다시 수행하게 된다. 이 때, 테스트 케이스들은 운용 환경에서 동작될 수 있도록 수정되어야 하는데, 테스트 케이스를 H/W의 특성에 따라 수정하거나 추가하는 작업이 필요하다. 그리고, 운용환경에서 동작하는 테스트 코드는 넉넉하지 않은 H/W 리소스를 효율적으로 사용해야 하며 실시간 동작하는 시스템을 효과적으로 테스트 할 수 있어야 한다

임베디드 S/W의 단위 기능을 수행하는 모듈은 하나의 바이너리 파일로 빌드(Build)되어 H/W에 탑재된다. 동일하게 임베디드 S/W 테스트를 위해 개발된 코드도 테스트하려는 어플리케이션이 코드 내에 포함되거나 분리되더라도 결국 하나의 바이너리 안에 포함된다. 운용환경에서 단위테스트를 수행하기 위해서는 실제 개발된 바이너리에 테스

트 코드를 삽입하고 컴파일과 링크를 다시 수행해야 하는 시간 소모적이고 번거로운 작업이 추가적으로 발생한다. 테스트 수행을 위해서 매번 테스트 케이스를 실제 코드와 함께 컴파일 하는 불편함은 운용환경에서의 테스트를 소극적으로 하게 하였다.

이에 본 논문에서는 임베디드 S/W 운용환경의 특성에 효과적으로 대응하여 컴파일 작업을 최초 1회로 최소화하면서도 테스트 케이스를 추가하거나 수정하여 테스트 할 수 있는 방법을 제안하려 한다. Data-Driven 기법이 적용된 테스트 모듈을 탑재하여 재 컴파일과 링크 없이 효과적으로 다양한 테스트를 동적으로 수행할 수 있게 해주는 방식이다. 그리고, 테스트 모듈을 자동 생성하고 범용적으로 사용할 수 있도록 설계하였다.

이 논문의 구성은 다음과 같다. 1절의 논문개요를 시작으로, 2절에는 임베디드 S/W 개발 환경과 테스트 이슈 그리고, S/W와 H/W 통합 관점에서 테스트 자동화의 중요성과 한계를 기술하고, 이어서, 자동화 기법이 적용된 효과적인 단위 테스트 기법을 제안할 것이다. 그리고, 3절에서는 테스트 기법을 실제 적용한 사례를 공유할 것이며, 마지막 4절에서는 결론 및 향후 방향을 제안할 것이다.

.

2. 본론

2-1. 임베디드 S/W 개발 환경과 테스트 이슈

임베디드 S/W는 주로 Windows, UNIX, LINUX 등의 개발환경(호스트)에서 개발된다. 그리고, 운용환경(타겟)의 특정 H/W와 임베디드 운영체제 상에 탑재되어 제품의 개발의 완료된다. 임베디드 시스템의 S/W는 정해진 표준과 규격 없이 어떤 목적을 갖고 설계된 H/W에서 동작하도록 개발되며, 개발 방식과 H/W의 기능을 고려한 테스트가 이루어진다. [1] 이러한 임베디드 S/W의 테스트는 일반적으로 그림 1과 같은 과정을 거친다. [2] 임베디드 S/W의 개발은 크로스 개발환경, 광범위한 실행 배치 아키텍처, 운용환경의 리소스 제약, 개발환경과 운용환경의 환경적 차이로 인해 다양한 제약이 발생한다. [3]

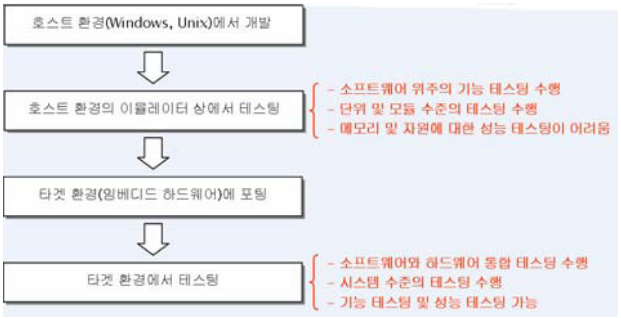

그림 1 임베디드 S/W 개발 및 테스트 과정

임베디드 시스템은 대부분의 경우에 H/W 및 S/W 개발이 동시에 진행되며 이러한 개발 특성으로 인해 H/W가 개발중인 시점에 S/W를 개발해야 하고 이를 위해 에뮬레이터(Emulator)와 같은 도구(Tool)를 사용하게 된다. 에뮬레이터는 S/W의 개발과 테스트 활동을 H/W와 독립적으로 진행될 수 있게 해주지만[4], 개발환경에서 이루어지는 테스트 활동만으로 임베디드 시스템의 기능 검증을 충분히 수행 했다고 말 할 수는 없을 것이다. 개발환경에서 단위 테스트나 모듈 테스트를 수행했다 하더라도 운용환경에서 최소한의 확인(Confirmation) 테스트라도 수행해야만 S/W의 품질에 대해 기본적인 신뢰를 가질 수 있는 것이다. 그렇기 때문에, 개발환경에서 구현된 테스트 케이스를 운용환경에서 실행해야 하는 일이 주요해 진다. 대체로 개발환경에서 구현한 테스트 드라이버를 운용환경에 맞게 포팅(Porting)하여 테스트한다. 개발환경에서 수행된 테스트를 운용환경에서 확인(Confirmation) 테스트하는 것을 H/W와 S/W 통합테스트라고 말한다. [5]

그런데, 개발환경의 테스트 드라이버를 운용환경에 맞게 포팅하는 것은 테스트 비용과 시간을 필요로 한다. 더 큰 문제는 제약이 존재하지 않도록 정확하게 동작 가능한 테스트 드라이버를 운용환경에 포팅한다는 것이 쉬운 일이 아니라는 것이다. 때로는 테스트 드라이버를 새로 개발하거나 테스트 케이스를 다시 설계해야 하는 상황이 발생하기

도 한다. 간혹, 직접 운용환경에 맞게 테스트 케이스를 설계하고 테스트 드라이버를 구현하는 방법으로 접근하기도 하지만, 이것 역시 많은 노력을 필요로 한다는 사실을 머지않아 발견할 수 있을 것이다.

2-2. S/W와 H/W 통합 테스트 관점에서 테스트 자동화의 중요성과 한계

테스트 자동화는 어떤 규칙적 특성을 갖는 기능이나 화면에 대하여 패턴을 도출하고 이를 테스트 케이스로 생성하여 반복적으로 테스트를 수행하는 것이다. 성공적인 테스트 자동화는 테스트 시간을 비약적으로 감소시킨다. 테스트 시간의 감소는 간과할 수 없는 관심거리다. S/W 개발 주기에서 테스트 단계는 명백히 존재한다. 하지만, 개발 일정에 의해 테스트 일정이 무시되는 경우가 종종 발생한다. 개발자와 테스터가 구별되지 않고 개발자가 테스트를 수행하는 경우라면 더욱 그러하다. 이러한 영향 속에서 테스트 활동의 큰 제약은 테스트 시간의 부족과 이로 인해 발생하는 비용 문제이다. 한정된 시간에 가능한 많은 테스트를 수행하려는 요구(Needs)는 꾸준하게 발생하고 있다. 테스트 자동화의 도입은 제한된 시간에 보다 많은 테스트 케이스를 수행할 수 있도록 해준다. 자동화된 테스트의 장점은 보다 많은 기능을 보다 짧은 시간에 테스트할 수 있도록 해주는 것이다. [6] 알려진 바에 따르면, 테스트 수행 시간은 매뉴얼 테스트와 자동화된 테스트 간에 큰 차이가 있다. 자동화된 테스트에 비해 매뉴얼(Manual) 테스트는 두 배 이상의 시간과 비용을 요구한다. [8]

자동화의 장점에도 불구하고, 임베디드 S/W는 복잡한 구조를 갖기 때문에 테스트 자동화에는 적합하지 않다는 것이 일반적이다. [9] 테스트 레벨 측면에서 보면, stub & 드라이버 기반의 테스트 활동은 단위 테스트를 의미하지만, 운용환경에서 수행하는 단위 테스트는 구조적으로 여러 모듈과 인터페이스 하고 모듈간의 메모리 공유가 기본조건이 되기 때문에 단위 테스트 활동과 시스템 혹은 통합 테스트의 경계가 모호해진다.

2-3. 자동화 기법을 적용한 단위 테스트 기법

우리는 앞서 여러 어려움과 한계를 함께 인식했다. 이제는 운용환경에서 검증할 수 있는 효과적인 단위 테스트 자동화 기법을 제시하여 이 어려움을 극복하고자 한다. 본 장에서는 테스트 드라이버를 운용환경에서 구동하기 위해 어떤 기능과 구조를 가져야 하는지, 그리고, 자동화를 위해 어떤 정보를 추출해야 하는지 설명할 것이다. 마지막으로 테스트 드라이버 생성 자동화 기법을 소개하겠다

테스트 드라이버는 시스템 및 시스템 컴포넌트를 테스트하는 환경의 일부분으로 테스트를 지원하는 목적 하에 생성된 코드와 데이터를 의미하며, 테스트 하네스(Harness)[7]와 같은 의미로 사용된다. 우리가 테스트 대상으로 하는 함수들은 입력 인자(Input Argument)와 출력 인자(Output Argument), 리턴값(return value)으로 인터페이스를 구성한다. 내부적으로는 전역변수(global variables), File System, DB 등을 사용하여 정보를 저장하거나 공유하게 된다. 단위 테스트는 내부 동작을 아는 화이트 박스(white box)기법과 내부 동작을 모르는 상태에서 수행하는 블랙 박스(black box)기법이 있다. 2가지 기법 모두 테스트 대상 함수를 호출할 때 입력 인자에 테스트 하려는 조건을 기반으로 데이터를 셋팅(Setting)하게 된다. 그리고, 리턴(return)이나 출력 인자를 통해 함수가 정상적인 기능을 수행하였는지 비교하여 테스트의 성공과 실패여부를 판단한다. [12][13]

테스트 자동화 도구를 설계할 때에는 도메인 엔지니어링을 통한 제품간의 공통성과 가변성을 추출하고, 가변화된 항목을 지원하기 위한 프레임워크를 제공하게 된다. [10] H/W 구성에 상관없이 가변항목이 플러그인 될 수 있는 프레임워크를 제공하는 구조를 기반으로 필요한 항목을 선택적으로 조합할 수 있도록 하는 것은 테스트 자동화 도구의

적용 범위를 확장할 수 있다. [11] 임베디드 S/W 단위 테스트를 자동화하기 위해 테스트 데이터와 테스트 드라이버를 분리하여 자동화 항목으로 도출하였다.

테스트 드라이버는 임베디드 S/W의 컴파일러들간의 컴파일 규칙의 차이와 임베디드 S/W 플랫폼에 포팅하기 위해 필요로 하는 컴포넌트의 특성에 대응할 수 있어야 하기 때문이다. 그리고, 테스트 데이터의 가변 항목은 일반적인 Primitive Type과 array, pointer로 구성된 구조체와 같은 자료 구조로 하였다.

테스트 드라이버의 자동 생성과 테스트 데이터의 종속성 제거를 위해 설계된 테스트 Module의 구조는 다음과 같다.

(1) 테스트 엔진

테스트 엔진(Test Engine)의 기능은 다음과 같다. 첫 번째는 테스트 대상이 되는 함수를 호출하는 함수 포인터 테이블(Function Pointer Table)의 관리 기능이다. 두 번째는 해당 함수의 입력인자/ 출력인자/ 리턴값으로 주고 받는 데이터의 구조를 해석(Parsing)하는 기능이다. 마지막으로 테스트 엔진은 호스트의 테스트 매니저로부터 프로파일(Profile)과 스크립트(Script)를 받아 해석하고, 테스트 드라이버로부터 결과를 받아 테스트 매니저로 전달하는 기능을 포함한다. 그리고, 테스트 엔진은 테스트 매니저에 의해 자동으로 생성된다.

(2) 테스트 드라이버

테스트 드라이버는 테스트 엔진에 의해서 인터페이스를 정의하고 있기 때문에 인터페이스만을 맞추어 주면 패턴화된 형태의 코드로 구현할 수 있다. 따라서 테스트 드라이버는 자동으로 생성된다. 테스트 드라이버는 테스트하려는 함수를 호출하는 기능과 호출 결과를 수집하는 기능을 포함한다.

(3) 테스트 매니저

테스트 엔진과 테스트 드라이버를 사전에 식별된 정보를 기반으로 자동으로 생성하는 기능을 포함한다. 테스트 엔진을 생성한 후 테스트 하려는 데이터와 함수를 식별하여 테스트 엔진에 프로파일과 스크립트로 전달하는 기능을 수행하고, 테스트 엔진이 테스트 드라이버를 이용해 테스트 데이터를 전달하고 그 결과를 수집하면 이를 테스트 매니저가 판단하게 된다.

테스트 데이터를 기존의 방법처럼 테스트 드라이버에 구현하게 되면 테스트 데이터를 변경할 때마다 컴파일을 다시 수행해서 링크 해주어야 한다. 하지만, 제안된 모듈은 테스트 엔진과 테스트 드라이버에 테스트 케이스를 위한 어떤 데이터도 구현하지 않는다. 그 대신 호스트와의 통신을 통해 테스트 드라이버가 필요로 하는 데이터를 테스트 엔진이 테스트 매니저로부터 전송 받아 함수의 구조에 맞게 해석하여 테스트 드라이버가 함수를 호출할 때 함께 전달하도록 한다. 이러한 구조로 인해 테스트 케이스를 추가하거나 테스트 데이터를 변경하는 것에 의해 테스트 엔진과 테스트 드라이버가 다시 컴파일 되어야 할 이유가 없어진다.

지금까지 설명한 테스트 엔진의 구조 및 테스트 드라이버, 호스트간의 흐름은 아래 그림과 같다.

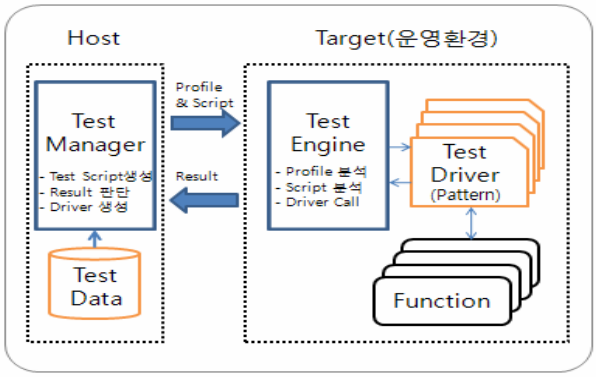

그림 2. 테스트 엔진과 테스트 드라이버, 호스트 구조

3 실제 임베디드 S/W 테스트에 적용한 사례

본 장에서는 앞서 제시한 방법을 실제 임베디드 S/W 테스트에 적용한 사례를 공유할 것이다.

각 임베디드 디바이스의 특성상 서로 다른 컴파일 환경, 구조, 그리고 서로 다른 통신 방식을 갖는

경우가 대부분이므로 테스트 매니저와 테스트 엔진간의 통신을 위한 포팅 작업이 필요하다.

통신 포팅이 완료된 상태에서, 테스트 하려는 함수를 대표적인 유형별로 10가지 식별하여 분석을 수행하였다. 함수의 분석은 헤더파일을 분석했는데, 명세서나 설계서를 통해 분석하는 것도 가능하다. 분석 방법과 상관없이 테스트 하려는 함수에 대하여 다음과 같은 정보를 식별하기만 하면 된다.

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

그림3. Printf, sprintf, fprintf 헤더

10개의 함수에 대하여 입력 인자/ 출력 인자/ 리턴 값의 구조와 함수의 이름을 식별했다면, 함수의 헤더는 테스트 드라이버에 자동으로 Include 된다. 식별한 정보를 기반으로테스트 엔진과 테스트 드라이버를 테스트 매니저를 이용해 자동 생성한다. 생성된 테스트 드라이버와 테스트 엔진을 운용환경인 타겟에 탑재하여 컴파일 한다. 여러 함수를 테스트하려 할 때 헤더의 include는 중복된 symbol의 redefinition으로 컴파일 error가 발생할 수 있기 때문에 필요하면 symbol을 정의한 공통 헤더를 제외한 다른 함수를 선언한 헤더를 하나로 묶어 별도의 헤더를 구현하는 기능이 필요하지만, 본 연구에서는 자동화 항목에서 제외하였다. 컴파일이 완료된 후 타겟에 이미지를 download한다.

호스트에서 정해진 테스트 데이터를 입력하여 타겟으로 전달한다. 테스트 엔진은 타겟으로 테스트하려는 함수 포인터 명과 함수의 입력으로 사용되는 값을 전달받아 해석한다. 그리고 해석된 정보를 테스트 드라이버에 전달한다. 테스트 드라이버는 이를 기반으로 해당 함수를 호출한다. 그런 다음, 출력 인자나 리턴 값과 같이 정의된 출력을 테스트 드라이버가 수집한 후 테스트 엔진에 전달한다. 테스트 엔진은 validation해야 할 정보를 테스트 드라이버로부터 전달받아 validation을 수행한 후 테스트의 성공(pass)/실패(fail)여부를 판단하여 호스트에 결과를 전송한다. 그림 2는 테스트 흐름을 보여준다.

이때, 사전에 설계된 테스트 데이터 이외에 다른 예외적인 테스트 데이터를 전달하기 위해서는 호스트에서 전달하려는 테스트 데이터의 값을 변경하여 호스트에서 타겟으로 전송하는 것만으로 해당 함수의 테스트를 수행할 수 있게 된다. 즉, 테스트 엔진은 추가적인 함수의 테스트를 결정하지 않는 한 컴파일을 다시 수행할 필요가 없다. 이로 인해 다양한 테스트 케이스에 대해서 테스트 드라이버의 수정 없이 외부에서 데이터를 전송하는 Data-Driven기법을 통해 단위 테스트의 커버리지(coverage)를 높이고 운용환경에서의 테스트 제약 중 하나인 제한된 시간 내에 생산성 있는 테스트 활동을 수행할 수 있게 된다.

아래 표1은 Data-Driven 기반의 테스트 방식을 적용하기 이전의 테스트 활동에 비해 향상된 장점이다.

	기존 Manual Test 방식	Data-Driven Test 방식	비고
Test 대상	10	10	
Test Case	100	100	
Test Driver	100	10	
Test Engine	0	1	
코드 구현	有	無	자동생성
Compile 회수	1~100	1	Test Case 추가에 따른 Compile 없음
Test Case 추가	10	10	
Test Case 추가시 Compile회수	1~10	0	

표1. Manual 방식 대비 Data-Driven 방식 장점

테스트 매니저에 의해 자동 생성되는 테스트 엔진과 테스트 드라이버는 범용적이다. 이번에 구현한 테스트 모듈은 테스트 대상이 되는 함수에 의존적이지 않다. 임베디드 C로 구현된 함수임을 전제하고, 새로운 함수를 테스트하기 위해서 테스트 드라이버와 테스트 엔진을 수정하지 않아도 된다.

4. 결론

룰 베이스 시스템(Rule-Based System)은 Parameter-Driven 기법을 개선하여 만들어진 전문가 시스템

이다. 룰 정보를 처리하는 룰 엔진을 기반으로 팩터(Factor)의 값이 변경되더라도 룰 엔진이 이를 식별하여 해당 로직에 적용하는 방식으로 S/W의 수정 없이도 비즈니스 로직을 변경할 수 있다. Data-driven기법은 이와 같은 원리로 함수의 입력 인자의 값을 factor로 도출하여 별도 처리하는 테스트 엔진을 생성하고, 테스트 드라이버를 테스트 데이터와 분리한 후 테스트 드라이버를 pattern화 하여 자동 생성한다. 이러한 단위 테스트 활동은 운용환경에서의 단위 테스트를 효과적으로 지원한다. 그리고, 시스템과 통합 테스트 활동으로 확장하여 임베디드 S/W가 갖는 운용환경에서의 테스트 제약을 해소하여, 생산성 있는 단위 테스트 활동을 통해 임베디드 S/W의 품질 이슈에 대응할 수 있게 해준다.

향후에는 테스트 수행과 연계하여 디버깅 정보를 추출하는 방법과 단위 테스트뿐 아니라 함수간 연동 테스트나 테스트 조건을 외부에서 동적으로 변경하는 방법으로 관련 연구가 확대되어야 할 것이다.

[참고문헌]

[1] Sol Shatz, Development of Distributed S/W: Concepts and Tools, Macmillan Publishing Company, 1993.

[2] 배현섭, “임베디드 테스팅 자동화 도구 및 요구사항”, 임베디드 월드, Vol.31., pp.88-93, July 2005.

[3] 김희진, “임베디드 S/W를 위한 테스트 드라이버 생성 자동화 기법 설계 및 구현”, pp.14-15 200702

[4] Bart Broekman and Edwin Notenboom, “Testing Embedded S/W”, Addison Wesley, 2003

[5] Harry Koehnemann, Dr. Timothy Lindquist, “Towards Target-Level Testing and Debugging Tools For Embedded S/W”, Arizona State University, 2002

[6] Paul C. Jorgensen, “S/W Testing A Craftsman’s Approach”, CRC Press

[7] Finkbine, R.B and N.A,. Kraft, “Introducing the 테스트 Harness: Automating the Test Suite.”, In Proceedings of the Information System Education Conference(SECON 2002”), San Antonio, Texas, USA, Nov.m, 2002

[8] Douglas Hoffman, “Using Test Oracles in Automation”, Pacific Northwest S/W Quality Conference, 2001.

[9] 장영숙, 여기대, 이현동, “Manual과 Automated Test에 대한 사례 연구”, 정보과학회 추계학술발표 논문집, 2003.

[10] 강교철, 이재준, 김병길, 김사중, “임베디드 시스템 개발 생산성의 획기적인 향상을 위한 S/W 제품 라인 공학”

[11] 천은정, 최병주, “테스트 프로세스 수행 도구의 설계 및 구현.”, 한국정보과학회논문지 : 컴퓨팅의 실제, 2004.

[12] Bart Broekman and Edwin Notenboom, “Testing Embedded S/W”, Addison Wesley, 2003.

[13] Kelvin Rodd, “Practical Guide to S/W System Testing.”,K.J. Ross & Associates Pty. Ltd, 1998.