관점지향 프로그래밍을 이용한 후크 기반의 임베디드 소프트웨어 테스팅

마영철*, 최윤희*, 최은만*
*동국대학교 컴퓨터공학과
e-mail:{randy, hataseyun, emchoi}@dongguk.edu

Hook-based Embedded Software Testing by using Aspect-Oriented Programming(AOP)

Young Chul Ma, Yun Hee Choi, Eun Man Choi*
*Dept of Computer Engineering, Dongguk University

요 약

임베디드 소프트웨어를 테스트하고 디버깅하려면 기능 분석, 프로세스 추적, 메모리 디버깅 등 다양한 기술들이 존재한다. 하지만 테스터가 임베디드 시스템 내부의 컴포넌트들의 사이에 결합을 발견하고 그 위치를 찾아야 하는 경우, 디버깅 기술과 도구만으로는 한계가 있다. 만약 테스터가 테스트 도구를 이용할 경우 이런 단점을 보완할 수 있지만 다양한 임베디드 시스템 환경에서는 테스트 환경만을 구축하는 데도 많은 노력과 시간이 소요된다. 따라서 이러한 문제 해결하기 위하여 본 논문에서는 관점 지향 프로그래밍(Aspect-Oriented Programming)을 사용한 후크(Hook) 개념을 적용하여 새로운 테스팅 아키텍처를 제안한다.

Key words : embedded software testing, software testability, hook mechanism, AOP.

1. 서론

임베디드 소프트웨어는 하드웨어와 밀접한 관련을 가지고 있다. 따라서 소프트웨어에 위험 및 오류가 포함되어 있을 경우 하드웨어에 오작동 및 결함을 야기할 수 있다. 임베디드 시스템에서는 이것이 리콜로 이어져 많은 경제적 손실을 야기 할 수 있다. 하지만 다양한 임베디드 시스템을 모두 확인하기에는 많은 시간과 비용을 요구하기 때문에 어려움이 있다.

따라서 임베디드 소프트웨어 개발자들은 소프트웨어의 결함을 찾기 위하여 디버깅 도구를 이용한다. 하지만 이러한 디버깅에는 다음과 같은 세 가지의 문제가 존재한다.

첫째, 개발자들은 디버깅 툴을 이용해 결함문제를 찾기 위해서 중단점을 설정하고 변수 값을 모니터링 해야 한다. 만약 예상했던 결과와 어긋날 경우 변수들의 값을 계속 확인해야 할 필요가 있기 때문에 버그를 찾을 때까지 많은 시간을 소비한다는 단점이 있다.

둘째, 일반적으로 임베디드 소프트웨어의 테스트를 효율적으로 진행하기 위해 에뮬레이터를 이용한다. 이는 실제 호스트 기기에서 테스트를 수행하는 것에 비해 에뮬레이터를 이용하면 비용을 줄일 수 있기 때문이다. 하지만 실제 테스트와 에뮬레이터를 이용한 테스트에는 차이가 존재한다.

셋째, 앞서 말한 에뮬레이터와 실제 디바이스 테스트를 통해 최종적으로 개발 호스트에서 다시 테스트를 실시해야한다. 이때 실제 장비에서 메모리 누수나 입/출력 오류 등의 문제가 생기면 해결하기가 어렵다.

본 논문은 이러한 문제의 해결을 위해 실제 디바이스에서의 테스트를 중심으로 다음과 같은 세 가지 기술을 제안한다.

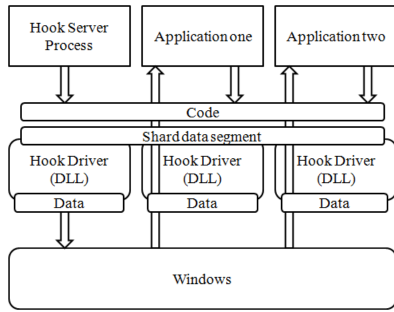
첫 번째, 모델의 성능을 측정하기 위하여 컴포넌트 테스트를 도입한다. 컴포넌트 테스트는 시스템 레벨 기능 테스트이며, 따라서 단위 테스트나 통합 테스트를 진행할 때 컴포넌트 단위로 진행하면 편리하다.

두 번째, 실제 장치에서 프로그램 테스트 할 때 외부 영향을 제거하기 위한 빌드인 테스트(Build-In Test)를 수행해야 한다. 임베디드 소프트웨어를 테스트 할 때 문제가 생긴다면 이것은 소프트웨어 자체의 문제이다. 시스템의 외부에서 CPU나 입/출력 기능 및 메모리 문제가 프로그램의 문제를 초래할 수 있다.

세 번째, 위 부분의 특징을 구현할 때 관점지향 프로그램 기술을 이용하면 보다 편리하게 테스트를 수행할 수 있다. 관점지향 프로그램을 이용하여 테스트 관심사를 분리할 수 있으며, 소프트웨어 내부 프로그램 로직을 수정하지 않고 공통의 관심사를 추출해 낼 수 있다. 제안한 기술의 타당성을 보이기 위하여 Aspect C++를 이용하여 홈메이드 큐 컴포넌트를 테스트 하였다.

2. 후크기반 테스트 개념

(그림 1)은 윈도우 운영체제에서의 후크 메커니즘을 나타낸다. 애플리케이션은 후크기능을 이용하여 다른 시스템의 메시지-핸들링 메커니즘을 감시할 수 있으며, 애플리케이션에 필터 함수를 추가하여 시스템과 프로세스 사이에 전달되는 메시지를 감시 및 처리할 수 있다.



(그림 1) 윈도우 후크 메커니즘

본 논문에서 제안한 메커니즘은 윈도우 운영체제에서의 후크 메커니즘과 비슷하지만 차이가 있다. 윈도우 후크 메커니즘은 시작 단계에서 개발과 테스트를 함께 시작한다. (그림 2)의 후크 아키택처를 이용하여 A, B, C의 컴포넌트를 구현하면, 테스트 도메인을 정할 수 있다. 개발자는 통합 과정을 진행하면서 테스트를 위해 컴포넌트 사이의 인터페이스를 파악해야 한다. 이 경우, 후크 메커니즘을 이용하여 컴포넌트 인터페이스를 관심사로 설정하여 테스트를 실행한다.

테스트를 실행할 때 컴포넌트 내부에서 함수 혹은 메서드가 호출될 경우 후크 메커니즘이 이를 추적할 수 있다. 뿐만 아니라 관련된 인자와 리턴 값도 확인할 수 있어 만약 잘못된 메서드를 실행되었을 때 후크 메커니즘은 문제를 확인하기 위하여 잘못된 메서드의 상세한 정보를 개발자한테 보여줄 수 있다.

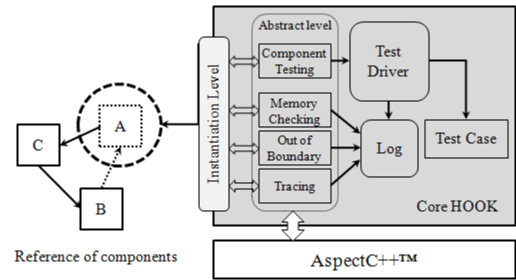
또한 컴포넌트 사이의 인터페이스를 통하여 컴포넌트가 이용될 경우의 호출 관계를 알 수 있다. 예를 들면, A 컴포넌트 내부에서 B 컴포넌트의 메서드 혹은 함수를 호출하면 A 컴포넌트 내부에 B의 어떤 함수가 호출되었는지 로그 문서 안에 저장한다.

이처럼 후크기반 테스트 메커니즘을 이용한다면, 단위 테스트부터 시스템 테스트까지의 테스트의 모든 단계에서 자세한 내역을 볼 있다. 그리고 개발자가 직접 실제 장치에서 테스트가 가능하기 때문에 결함이나 문제를 발견하면 곧바로 해결할 수 있다. 즉 에뮬레이터에서 테스터하고 또 다시 테스트를 필요 없다.

테스터는 원시코드를 수정하지 않고도 후크 메커니즘을 이용해 테스트 케이스와 오라클, 테스트 드라이버를 쉽게 조작할 수 있다. 테스트 종료 후 테스트 케이스의 삭제도 편리하다. 또한 테스트 결과는 테스트 작업하는 즉시 확인할 수 있고 공통된 양식으로 로그 파일을 저장 가능하다.

3. 후크기반 테스트 아키텍처

후크기반에 테스트 아키텍처는 2단계로 나누어진다. (그림 2)의 화이트 박스 부분은 본 논문에서 제안한 후크 기반 테스트 아키텍처의 핵심이다. 이 아키텍처를 기반으로 테스트 케이스의 모든 테스트 기능과 부가 기능들을 구축할 수 있다. 일반적으로 임베디드 기반 소프트웨어에 많이 쓰는 프로그램 언어는 C/C++로 개발 환경에 맞추기 위해 AspectC++ 컴파일러를 이용하여 후크기반 테스트 아키텍처를 구현하였다.



(그림 2) 후크 아키택처

```
class Queue{
  private:
    QNode *q_pFront;
    QNode *q_pRear;
    int q_size;
  public:
    Queue(){ ...// constructor
      pTemp = (QNode *)malloc(sizeof(QNode));
    }
    ~Queue(){...// destructor
      while(NULL != pTemp ){
      free(pTemp);
      }
    }
    bool push(const T&t){
     ...//
      pTemp = (QNode *)malloc(sizeof(QNode));
    }
    T pop(){
      if(empty()){ throw error<T>("Overflow"); }
      ...
      free(pTemp);
      return t;
    }
    const boolisEmpty()const{...}
    const intsize()const{...}
    T getFront(){...}
};
```

(그림 3) C++ 언어 구현된 큐 컴포넌트 예제

(그림 2)의 그레이 박스 부분은 본 논문 후크 테스트 핵심 부분으로 크게 세 가지 영역으로 나눌 수 있다. 이 영역들은 테스트 드라이버 모듈과 테스트 케이스 모듈, 로그 모듈로 구성되었다. 테스트 드라이버 모듈은 테스트 케이스와 테스트 이벤트를 구동을 시키는 모듈이다. 테스트 케이스 모듈은 테스트에 필요한 데이터와 데이터베이스 생성을 관리하는 모듈이다. 마지막 로그 모듈은 테스트 과정 중, 혹은 테스트 후의 결과를 저장을 시키는 모듈이다.

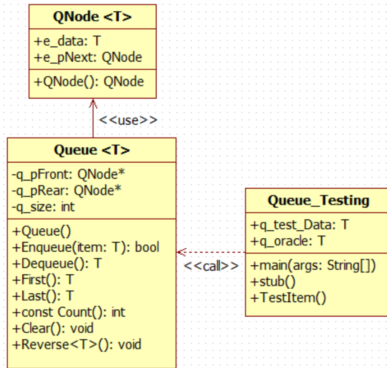
후크 테스트 기능은 추상 레벨로 네 가지의 기능을 가지고 있다. 그 중 컴포넌트 테스팅 부분은 후크기반 테스트 아키텍처의 핵심부분으로 블랙박스 테스트, 통합 테스트를 모두 포함하고 있다. C/C++ 언어 프로그램에서는 메모리 문제를 피할 수 없다. C/C++ 언어 중에 가비지 수집 처리한 메커니즘이 없으므로 개발자들이 할당된 공간을 전부 수동으로 반납해야한다. 만약 문제가 발생하면 어떤 부분에서 메모리 누수가 일어났는지 한 번에 찾기는 어렵다. 따라서 메모리 문제를 자동 해결할 수가 없지만 메모리 체킹 기능을 이용하여 메모리 문제 발생 위치를 알 수 있다. 함수나 메서드 호출이 될 경우 메모리 영역을 벗어나는 부분은 전부 로그 안에 기록할 수 있다. 이러한 추적 기능은 네 가지 기능 중에 가장 유용한 기능으로 테스트 때 실행 순서, 호출 관계 등 문제를 해결하기 위해 도움이 될 수 있는 정보를 알아낼 수 있다.

(그림 3)은 후크 기반 테스트 아키텍처를 시험하기 위한 예제로 홈-메이드(home-made) 큐 컴포넌트로 다음 장에서 테스트 방법에 대해 자세히 서술한다.

4. 후크기반 임베디드 테스트

4.1 일반적인 컴포넌트 테스트

이 장에서 후크 테스트 방법과 일반적인 테스트 방법의 비교를 통해 기능을 자세히 알아본다.



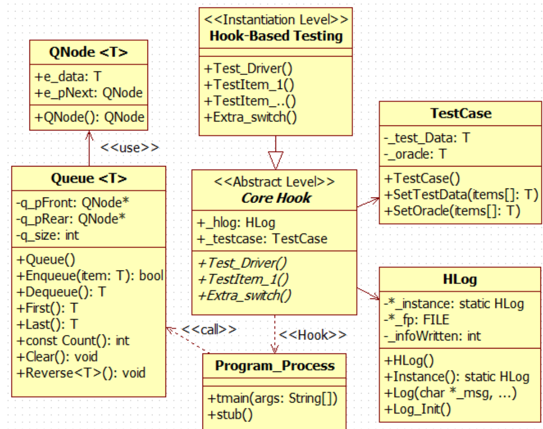
(그림 4) 일반적인 테스트 방법

홈-메이드 큐 컴포넌트를 이해하기 위해 UML을 통해 구조를 알아본다. (그림 4)는 일반적인 컴포넌트 테스트 방법, (그림 5)는 후크 테스트 방법이다. 일반적인 컴포넌트 방법은 테스트를 시작하기 전 테스트 케이스를 먼저 초기화 하고 매개변수를 변수에 저장한다. 그 후 순차적으로 테스트를 실행하면서 테스트 데이터와 결과를 데이터베이스와 비교한다. 반복적으로 테스트를 진행하며 테스트 함수를 완성할 때까지 진행한다. 시험 순서는 다음 알고리즘과 같다.

```
load initialize.
   forall   d ∈ Test Data Sets
                  Put the d into q_test_Data array as sequence.
   forall   o ∈ Oracle Sets
                  Put the o into q_oracle array as sequence.
   forall   f ∈ Functions
            Invoke the functions under test in TestItem list.
   Begin
            d ← q_test_Data0
            o  ← q_oracle0
            for each d in the Test Data Sets (Length(TDS) ≥1)
                 if f  is invoked in TestItem  then
                      Put d as parameters into f.
                 if f return a valuethen
                      Compare with relevanto
                 ifo equal  f(d) then
                      This test case can passed test.
 end
 forall f(d) equal o
                  This component passed the testing.
```

4.2 후크기반 컴포넌트 테스트

일반적인 컴포넌트 테스트 방법은 다음과 같은 단점이 있다. 첫째, 테스트 코드를 원시코드에 삽입한다는 것이다. 만약 테스트 코드를 헤더 파일에 추가한 경우 테스트가 끝난 후 삽입된 테스트 코드를 하나씩 원시코드에서 삭제해야 한다. 둘째, 테스트의 결과를 확인하기 위하여 출력함수를 많이 이용해야 한다는 것이다. 이는 상당히 번거로운 작업이다. 셋째, 디바이스에서 소프트웨어를 실행할 경우 외부에서 발생한 문제들을 확인할 수 없다. 그 중에서도 소프트웨어에서 메모리 문제를 발생할 경우에는 다른 도구를 이용해 확인해야 한다.



(그림 5) 후크기반 테스트 기법

이와 같은 문제들은 후크 테스트 기법을 이용하면 문제를 해결할 수 있다. 먼저 테스트 코드를 애스펙트 컴파일러를 이용하여 컴파일러 한다. 이렇게 하면 테스트 대상 원시코드와 직조되어 테스트가 실행된다.

이 방법은 원시코드를 수정하지 않고 직조하는 방법으로 원시코드에 테스트 코드를 쉽게 직조, 삭제할 수 있다. 테스트의 중간 결과를 확인을 원한다면 애스펙트 반사(reflection) 메커니즘을 이용한다. 결과 로그는 단일객체로 로그의 객체를 싱글턴 패턴을 이용해 접근하고 로그를 기록한다. 추상 후크 테스트 코드는 다음과 같다.

```
aspect Core_Hook {
   pointcut   virtual Component() = 0;
   pointcut   virtual ProcessMethod() = 0;
   pointcut   virtual IOMethod() = 0;
   pointcut   virtual Test_Driver() = 0;
   HLog     hlog = HLog::Instance();
   TestCase    testCase;

   advice Component() : slice    class{ };
   //constructed function.
   pointcut constr() = construction(Component());
   advice constr() : after(){
       hlog.Log("%s initialization", JoinPoint::signature());
       testCase.SetTestData();
       testCase.SetOracle();
       Test_Driver();
   }
   //no return values or parameters.
   pointcut proces()    = call(ProcessMethod())
                      && within(Core_Hook);
   advice process() : after(){
        hlog.Log("%s be invoked\n",JoinPoint::signature());
    }
    //have return values and parameters.
    pointcut iometh() = call(IOMethod()) && within(Core_Hook);
   advice iometh () : before(){
      hlog.Log("before %s",JoinPoint::signature());
      for(int i = 0; i<JoinPoint::args(); i++)
          hlog.Log("arg(%d) : %d\n",i,tjp->arg(i));
   }
   advice iometh () : after(){
      hlog.Log("after %s",JoinPoint::signature());
      hlog.Log("Result : %s\n",tjp->result());
   }
 };
```

5. 후크기반 아키텍처를 이용한 테스트 결과와 비교

본 장에서는 후크 테스트 기법을 이용한 블랙박스 테스트의 결과를 밝힌다. 또한 일반적인 테스트 방법에서 많이 사용하는 MEMWATCH 메모리 테스트 도구와 비교하여 장, 단점을 기술한다.

5.1 후크기반 큐 테스트 결과

앞서 제시한 (그림 4)의 일반적인 테스트 방법은 다음과 같이 테스트를 한다. 테스트의 종류는 세 가지로 큐의 구조함수, 인자와 리턴 값이 없는 Clear() 함수, 그리고 인자나 리턴 값이 있는 나머지는 함수들이 있다. 따라서 (그림 5)에 이 세 가지 중류의 함수에 대응되는 세 가지 충고(advise)가 있다. constr() 충고를 활성화 하면 큐의 객체를 생성할 때 이 충고를 이용하여 생성내용을 기록할 수 있다. process() 충고는 인자와 리턴 값이 없는 함수 호출 때 기록하기 위한 충고이다. iometh() 충고는 인자나 리턴 값이 있는 경우에 테스트 내용을 기록하기 위한 부분이다. (그림 6)을 통하여 결과를 알 수 있다.

```
=== HookTesting  Copyright (C) 2010    Randy Ma ===
Started at Sun Aug  8 21:29:47 2010

... ...
before "bool Queue::Enqueue(int)"  arg(0) : 3
after "bool Queue::Enqueue(int)"  Result :

before "bool Queue::Enqueue(int)"  arg(0) : 8
after "bool Queue::Enqueue(int)"  Result :

before "bool Queue::Enqueue(int)"  arg(0) : 5
after "bool Queue::Enqueue(int)"  Result :

before "int Queue::Dequeue()"
before "bool Queue::empty()"
after "bool Queue::empty()"  Result :
after "int Queue::Dequeue()"  Result : 3
before "int Queue::Dequeue()"
before "bool Queue::empty()"
after "bool Queue::empty()"  Result :
after "int Queue::Dequeue()"  Result : 8

before "int Queue::First()"
before "bool Queue::empty()"
after "bool Queue::empty()" Result :
after "int Queue::First()"  Result : 5

before "int Queue::Dequeue()"
before "bool Queue::empty()"
after "bool Queue::empty()"  Result :
after "int Queue::Dequeue()"  Result : 5
before "bool Queue::empty()"
after "bool Queue::empty()"  Result :
... ...
```

(그림 6) 후크 이용한 테스트 결과

5.2 MEMWATCH 2.71 Vs 메모리 체킹

MEMWATCH은 C언어 오픈소스 메모리 테스트 도구이다. 이 도구는 메모리 누수를 검사할 뿐만 아니라 메모리 더블프리(double free)와 오버플로우 문제를 검색할 수 있는 기능도 포함되어 있다. 이 도구를 이용하여 메모리 누수를 확인하고 본 논문에서 제시한 후크기반 테스트 결과와 비교했다.

<표 1> 메모리 체킹 과 MEMWATCH 비교

Program name	Language	Loc	Large	Unfree	Total
Memory Checking	AspectC++	117	210 bytes	20 bytes	730 bytes
MEMWATCH	C	2,224	220 bytes	20 bytes	730 bytes

<표 1>을 통해 알 수 있듯이, 보통 MEMWATCH를 이용하면 2224개의 라인으로 구현된 메모리 누수 확인 기능이 후크기반 테스트 기술을 이용하면 117라인으로 그 비교 값이 현저히 차이가 나는 것을 확인할 수 있다.

결과를 통하여 출력한 내용이 같지만 할당된 최대 메모리 값의 차이를 알 수 있다. 예제 코드 최대 할당된 메모리 값이 210바이트인데 MEMWATCH 결과 최대 220바이트로 차이가 있는 것을 확인할 수 있다. 이는 성능 면에서 후크 기반 아키텍처를 이용한 테스트가 낫다는 것을 확인할 수 있었다.

6. 결론

본 논문에서 제안한 후크기반 테스트 기법을 이용하여 실험한 결과를 통해 실제 임베디드 소프트웨어에서 테스트 실행이 가능하며 후크가 자유자재로 탈부착 가능함을 확인하였다. 구현된 기능을 간단하지만 테스트 시 필요한 결과를 충분히 가지고 있다.

향후 연구 과제로는 후크 기능의 자동 생성 가능성에 대한 것이다. 추상 후크가 아닌 충고는 매번 새로 작성되어야 하는데 이 부분을 자동화 한다면 대규모 임베디드 시스템에서 추적과 디버깅이 더 효율적으로 이루어질 수 있다.

참고문헌

[1] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum, *Building Embedded Linux System*, 2nd edition, O'REILLY, 2008.

[2] Peter M.Kruse, Joachim Wegener, Stenfan Wappler, "A Highly Configurable Test System for Evolutionary Black-Box Testing of Embedded Systems," *ACM GECCO'09*, Montreal, Canada, pp.1545-1552, 2009.

[3] Yichen WANG, Zhenzhen ZHOU, "Software BIT Design and Testing for Embedded Software," Reliability,Maintainablity and Safety, 2009. ICRMS 2009. 8th International Conference, pp.703-707, 2009.

[4] G. Kizales, J.Lamping, A.Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, "Aspect-Oriented Programming," *Proc. of the 11th European Conference Object-Oriented Programming. LNCS1241*, pp.220-242, 1997.

[5] Hamid Mcheick, Aymen Sioud, "Comparison of Garbage Collector Prototypes for C++ Application," Computer System and Application ,2009. AICCSA 2009. IEEE/ACS International Conference, Canada, pp.668-674.

[6] JM Bruel, J Araújo, A Moreira, A Royer, "Using Aspects to Develop Built-In Tests for Components," *AOSD Modeling with UML Workshop at UML'03*, San Francisco, USA, 2003.

[7] GmbH, Ma tthias Urban, and Olaf Spinczyk, *Documentation AspectC++ Language Reference*, Pure-systems, 2006.

[8] Bruel, J.-M., and Royer, "A. Aspects and BIT: a comparative case study,". UPPA Technical Report 2003-07-01, 2003.

[9] Jean-Marc Jezequel, Noel Plouzeau, Torben Weis, and Kurt Geihs, "From Contracts to Aspects in UML Designs," in Proceddings of the Workshop on Aspect-Oriented Modeling with UML at AOSD' 2002.

[10] AspectC++ project homepage http://www.aspectc.org

[11] T.Kishi, and N.Noda, "Aspect-oriented context modeling for embedded systems", *Early-Aspects 2004*, 2004.