동적 인스트루멘테이션 기법을 이용한 임베디드 S/W 테스팅 프레임워크

정도균*, 신석규**
S/W시험인증팀, TTA 시험인증연구소*,**
e-mail:{dkjeong*, skshin**}@tta.or.kr

The embedded software testing Framework using dynamic instrumentation techniques

Doe-Kyun Jeong*, Seok-Kyoo Shin**
Telecommunications Technology Association, Software Quality Evaluation Team*,**

요 약

최근 임베디드 시스템 기술은 마이크로프로세스의 저가, 소형화, 고성능화에 따라 제품 경쟁력의 핵심이 H/W 생산기술에서 S/W 시험·품질보증기술로 이동하고 있다. 그러나 마켓-플레이스의 다양한 임베디드 S/W 아키텍처 및 개발환경은 임베디드 S/W 자동화 시험방법에 대한 제약을 지닌다. 이에 본 논문에서는 동적 인스트루멘테이션(Dynamic Instrumentation)을 기법 및 그에 대한 디자인 프레임워크(Design Framework)를 제시한다. 이러한 동적 인스트루멘테이션 기법을 적용한 프레임워크는 다양한 임베디드 시스템 환경을 추상화(Abstraction)하여 임베디드 S/W 시스템의 실시간 수행과정에서 고품질달성과 관련된 데이터를 수집하고 분석하는데 이용되어질 수 있다.

1. 서론

임베디드 S/W는 일반 S/W와는 달리 실시간성, 고신뢰성, 저전력성을 요구하는 특성을 가지며 통상적으로 타겟(Target) 플랫폼이 다른 교차 개발환경(Cross-Platform Development Environment)으로 구성된다.[1] 이러한 임베디드 S/W 시스템 특성으로 일반 소프트웨어의 블랙박스 테스팅 또는 화이트박스 테스팅 기법을 일반적으로 적용하는데 많은 제약사항을 가진다. [1]

또한, 임베디드 환경에서는 데스크톱 PC나 서버환경의 S/W처럼 치명적 오류발생으로 인한 리부팅은 있을 수 없는 일이고 H/W 여분(Hardware Redundancy)을 통해 안정성을 높이는 것도 매우 어렵다. 즉 검증된 코드가 아니거나 검증되지 않은 임베디드 S/W 시스템이라면 안 되는 것이다. [2]

따라서, 임베디드 S/W는 데스크톱 PC나 서버환경에서 수행되어지는 패키지 S/W와는 다른 전략과 방법으로 개발되어야 하며 품질 요구사항(Time Requirement, Efficiency Requirement, Specification Requirement등[1])을 만족하기 위해서는 전체 S/W의 개발 생명주기에서 단계적인 S/W정확성 검증과 타당성 검사에서도 다른 방법을 요구하게 된다.

이에 임베디드 S/W시스템의 다양한 교차-개발환경 및 플랫폼 제약사항을 극복하고 S/W 요구사항을 검증할 수 있는 테스팅 기법 및 도구가 필요하다. [2]

본 논문은 이러한 임베디드 S/W 시스템 특유의 플랫폼 의존적 테스팅 기법 및 도구의 문제점에 주목하고 이러한 제약사항을 극복하기 위한 방안으로 소스 코드 인스트루멘테이션(Source Code Instrumentation) 기법[3]을 이용, 임베디드 S/W에 대한 품질 요구사항을 실시간 운용과정에서 검증할 수 있는 동적 인스트루멘테이션 프레임워크(Dynamic-Instrumentation Framework)에 대해 연

구하였다.

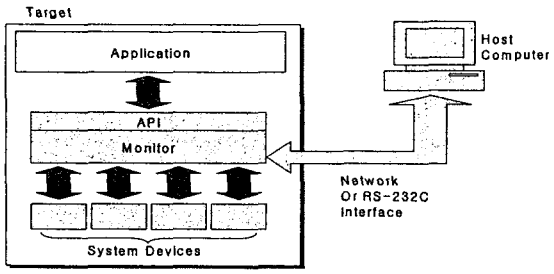
본 논문은 총 4장으로 구성된다. 2장에서는 본 논문과 관련된 연구에 대해 알아보고, 3장에서는 동적 인스트루멘테이션 기법을 적용한 임베디드 S/W 테스팅 프레임워크를 제시하고 설명한다. 마지막으로 4장에서는 결론 및 향후 연구방향에 대해 논의한다.

2. 관련연구

본 장에서는 임베디드 시스템의 교차-개발환경, 시험 제약사항, 소스코드 인스트루멘테이션 기법, 실시간 운용 S/W의 품질관련 데이터 수집과 관련한 프로파일링 기법(Profiling Technique)에 대해 알아본다.

2.1 임베디드 S/W의 교차-개발환경

임베디드 S/W 시스템은 통상적으로 개발 플랫폼과 S/W가 운용되어지는 타겟 플랫폼이 일치하지 않는다. 이러한 개발환경[그림 1]을 교차-개발환경이라고 한다.



[그림 1] 교차-개발환경

이러한 임베디드 S/W 타겟 시스템은 네이티브(Native) 개발환경과는 달리 개발환경을 실행할만한 능력이 없는 경우가 대부분이기 때문에 [그림 1]과 같이 개발호스트와 타겟시스템을 따로 구성하여 임베디드 S/W를 개발하는 것이 일반적이다.

2.2. 임베디드 S/W 시험 제약사항

임베디드 S/W 시스템의 교차-개발환경을 구성하는 타겟 시스템은 다음 [표 1]과 같이 플랫폼이 매우 다양하고 복잡한 RTOS/Chip벤더들의 조합들로 구성된다. 또한, 임베디드 시스템 설계·제조업체가 내부의 사용용도로 개발한 독자(Proprietary) OS가 임베디드OS 시장의 반 이상을 차지하고 있다. [2]

구분	종류
개발 호스트 플랫폼	윈도우2K, 윈도우95/98, 솔라리스, 리눅스, HP-UX, IBM-AIX .....
실행 타겟 플랫폼	Vxworks, pSos, Nuclueus, Plus, Lynx/Lynux, QNX, 윈도우CE, 임베디드 리눅스, VRTX, 솔라리스OS-9 Chorus, Inferno, TeaPos .....

[표 1] 대표적인 호스트 및 타겟 플랫폼 환경

따라서 리얼타임, 임베디드 S/W의 테스팅 기법 및 도구는 되도록이면 많은 타겟 시스템 환경을 지원하고 시험 가능하여야 한다. 범용 컴퓨터 개발환경의 경우, 개발 호스트 시스템이 곧 타겟 시스템이지만 임베디드 S/W환경에서는 개발 호스트 플랫폼과 실제 S/W가 실행·운영되는 타겟 플랫폼이 구분되어 존재하고 그 종류가 매우 다양하기 때문이다.

이러한 이유로, 교차-개발환경을 기반으로 한 임베디드 S/W개발·시험방법론 및 도구는 일반 S/W와의 차이를 보여준다. 또한, 임베디드 S/W시스템에서 제공하는 개발환경은 특정 교차-개발환경만을 지원하며 디버거(Debugger) 또는 H/W기반의 트레이서(Tracer) 역시 풍부한 H/W지식과 시스템 S/W의 개발경험을 요구하고 이를 테스팅·검증 시스템으로 사용하기에 어려움이 따른다. [2]

다양한 규격과 이기종이 존재하는 임베디드 S/W 개발환경 및 특성과 테스팅(및 도구를 개발·적용)에 대한 제약사항은 다음 [표 2]와 같다.

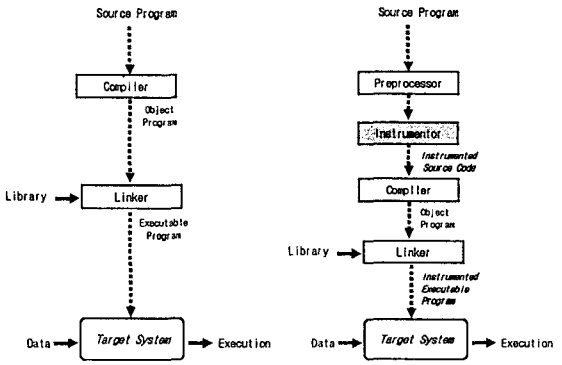
특성	테스팅 (도구) 제약사항
■ 실행시스템 용도에 따라 실시간 처리 요구 ■ 고도의 신뢰성 요구 ■ H/W에 최적화된 S/W 기술 요구 ■ 특정시스템의 실행을 목적으로 개발 ■ 네트워크 및 멀티미디어 처리기능 요구 ■ 교차-개발환경 ■ 다양한 RTOS/Chip벤더	■ 타겟시스템 기반 테스팅의 어려움 ■ 대분의 시험은 시뮬레이터, 에뮬레이터에서 수행 ■ H/W 및 S/W에 대한 고도의 지식을 요구함 ■ 플랫폼에 의존적인 디자인의 한계로 자동화 시험도구의 적용 유연성이 낮으며 새로운 임베디드 시스템 또는 Product-Line에 대해서는 새로운 시험도구를 필요로 함 ■ 그레이 박스(Gray-Box) 테스팅 요구사항이 많음

[표 2] 임베디드 S/W의 특성 및 시험 제약사항

2.3 소스코드 인스트루멘테이션(Source Code Instrumentation)과 프로파일링(Profiling) 기법

소스코드 인스트루멘테이션(SCI)란 대상이 되는 소스파일 내에 특정코드를 원하는 위치에 삽입하는 기술을 말한다.[2] 자동화되어진 SCI 기법은 일반적으로 빌드프로세스(Build Process)에서 프리-프로세싱(Preprocessing)된 네이티브(Native)코드에 대해 수행되어지고 기존 빌드프로세스에 새로운 인스트루멘테이션 단계가 추가된다. [3]

다음 [그림 3]은 SCI적용 이전·후의 빌드프로세스 간 차이점을 보여준다.



[그림 3] SIC이전·이후의 빌드-프로세스 비교

이러한 SCI의 자동화 기법을 이용하면 컴파일 타임 시 임베디드 S/W 품질관련 데이터를 수집하는 특정 코드를 자동으로 삽입할 수 있다. 다음 [그림 4]는 인스트루멘테이션 사례를 보여준다.

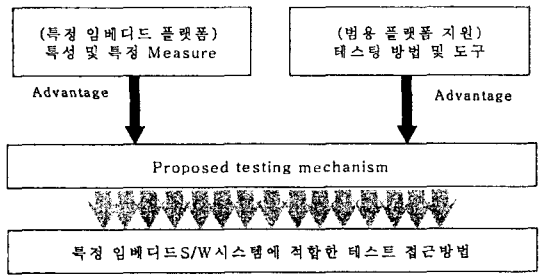
```
while( new_state != NONE_S )
{
  .....
  switch ( state ) {
     case ENTER_S:
     // Instrumentation Code for Performance Profiling
     // EventID, EventType, 호출된 시스템시간 변수에 저장
     ui_get_keys(FALSE);
     // Instrumentation Code for Performance Profiling
     // EventID, EventType, 호출된 시스템시간 변수에 저장
     if( /* Instrumentation Code for Coverage Profiling */
         /* EventID, EventType, Flag을 지정변수에 저장 */  )
      {
        .....
      }
  ...........
```

[그림 4] In-Line Instrumentation Code의 사례

이러한 Instrumented Code의 Compiled Code는 타겟 시스템에서 실행될 때, 하드웨어 아키텍처 디자인과 일치하는지에 대한 동적 행위(Dynamic Behavior)의 흐름분석(Flow Analysis), 프로그램 실행동안의 복잡한 입·출력, Function calls/loop iterations/conditionals등을 포함한 소스코드 테스트 커버리지, 동적 메모리 사용 프로파일링(Dynamic Memory Usage Profiling)[4]과 같이 임베디드 S/W 시스템의 실시간 운용과정에서 발생하는 추상화된 고수준(High-Level) 데이터를 획득·분석하는 프로파일링 기법에 이용할 수 있다.

3. 연구결과

본 장에서는 다양한 임베디드 S/W시스템 플랫폼을 추상화(Abstraction)하기 위한 동적 인스트루멘테이션 기법과 그와 관련된 디자인 프레임워크(Design Framework)를 제시한다. 이러한 프레임워크는 임베디드 S/W시스템 특성과 관련된 특정 측정(Measure)과 일반적인 시스템에 적용 가능한 테스트항목들로부터 특정 임베디드 교차-개발환경에 적합한 테스트 방법을 어셈블링 하는 메커니즘이다. [그림 5]



[그림 5] 제안 메커니즘

3.1 테스팅 프레임워크 개요

제시하는 디자인 프레임워크의 중요 이슈는 타겟 플랫폼에 대한 컴파일러(Compiler), 링커(Linker), 운영체제 라이브러리(Library)등의 빌드환경(Build Environment)과 타겟-호스트시스템간의 통신채널 등을 동적으로 인스턴스화(Initialization)함으로써 특정 임베디드 S/W시스템의 교차-개발환경(RTOS/Chip)에 대응할 수 있는 유연한 테스팅 도구 아키텍처의 설계이다.

3.2 타겟 시스템의 요구사항

제시되는 디자인 프레임워크를 정의하기 이전에, 적용 가능한 임베디드 S/W의 타겟 시스템의 최소한의 요구사항은 다음 [표 3]과 같다. 이러한 사항은 임베디드 S/W 테스팅 프레임워크에서 적용코자 하는 테

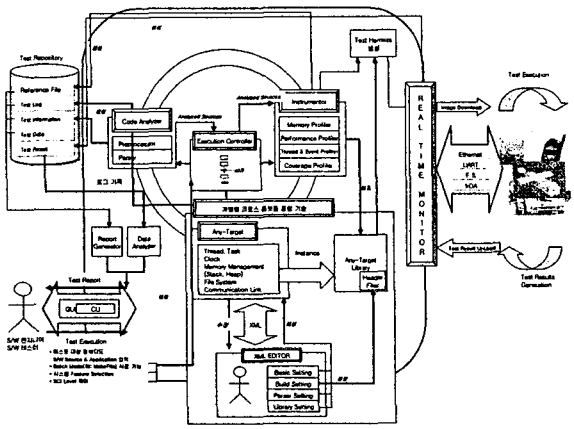
스팅을 수행하기 위해 사전에 타겟에서 구성·구현되어야 한다.

구 분	Coverage Profiling	Thread & Event Profiling	Memory Profiling	Performance Profiling
데이터 추출 능력	O	O	O	O
데이터 저장 능력	O	O	O	O
스택 공간	O	O	O	O
힙 관리 기능	·	·	O	·
Task 관리기능	·	O	O	O
Private Data	·	O	O	O
클럭 인터페이스	·	·	·	O
호스트시스템과의 링크속도	·	√ (For Realtime)	·	·
Task 관리기능	·	O	O	O

[표 3] 타겟시스템 요구사항

3.3 테스팅 프레임워크 구성

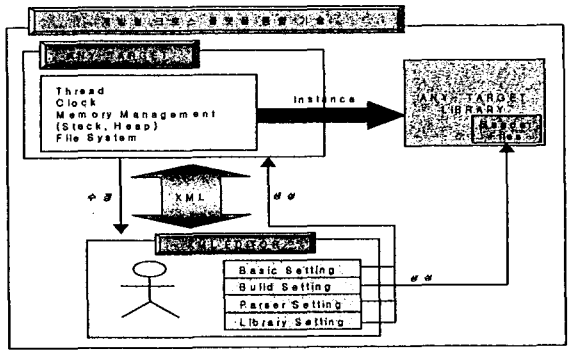
이러한 디자인 프레임워크는 [그림 6]처럼 크게 4개의 주요 컴포넌트로(Any-Target[1](Target Building Environment Library Component), Instrumentor[2](Instrument Component), Execution Controller Component[3], Code Analysis Component[4])로 구성된다. 프로파일링에 의해 생성되는 결과데이터 또는 원시데이터를 호스트시스템으로 업로드 하는 RealTime Monitor 및 분석방법은 본 논문에서는 언급하지 않도록 한다.



[그림 6] 프레임워크(Framework)

· Any-Target[1] : 해당 교차-개발환경의 컴파일러가 인식하는 동적 메모리 사용, 코드 커버리지, 성능 프로파일링 관련 Library를 포함한다.

이러한 Library는 타겟 시스템 기반의 임베디드 소프트웨어 개발 환경을 차용해 Testing을 위해 Instrumentation된 소스코드에서 사용되는 Testing Library를 독립적인 형태로 구성하여 운영한다. 다음 [그림 7], [표 3]은 Any Target Component의 구성을 나타낸다.

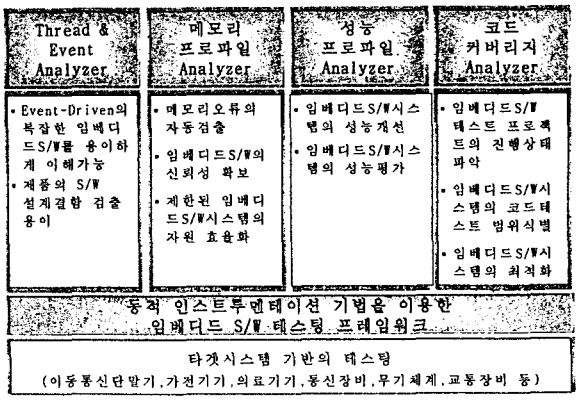


[그림 7] Any-Target 구성도

구분	내용
기본설정	개발환경에서 사용되어지는 파일 확장자명, 기본 플래그, 환경 변수, 일반 변수등을 기술
빌드설정	빌드 프로세스에서 요구되는 함수들을 구성. 본 항목에는 전처리, 컴파일, 링크, 실행 및 디버깅 정보를 포함.
라이브러리설정	Target System Library의 사용을 기술.
파싱설정	개발언어에서 제공하는 키워드, 예약어, 토큰(TOKEN)을 설정

[표 3] Any-Target 구성 요소

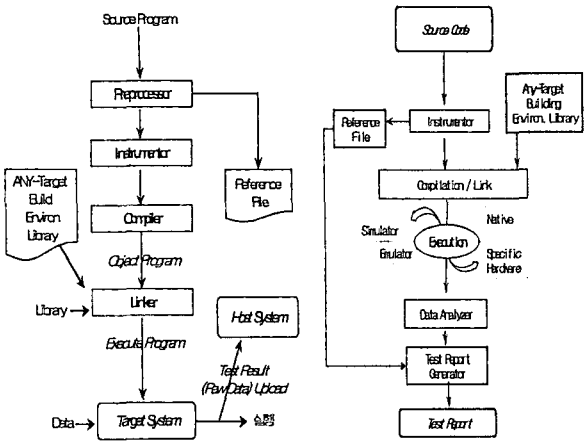
· Instrumentor[2] : 테스트 하니스(Test Harness) 생성대상이 되는 소스파일 내에 Any-Target에서 동적으로 생성된 특정코드(테스팅관련 Library, 분석관련 Library)를 자동으로 삽입하여 임베디드 S/W시스템의 운용 중, 실시간 프로파일링을 수행하는 모듈이다. 일반적으로 Instrumentation기법을 이용하여 임베디드S/W시스템에 적용되어질 수 있는 프로파일링 종류와 그 기대효과는 다음 [그림 8]과 같다. [1][3][4]



[그림 8] 인스트루멘테이션 기법을 이용한 프로파일링 종류와 기대효과

추가되어지는 코드는 기존의 개발환경에서 완벽하게 인식 가능한 형태로, 식별자(Identifier) 및 Testing Library를 추가시키거나 기존의 Native Library를 Testing Library 호출로 대체시킨다. [그림 8]

이러한 Library 및 식별자의 구현은 Any-Target Component에서 구성된다. 이때 생성되는 Reference File은 프로파일링 기법에 의해 생성된 원시데이터와 조합하여 Test Report를 생성하기 위해 참조되는 파일이다.



[그림 8] Instrumentation Procedure

· Execution Controller Component[3] : 타겟시스템 기반의 테스팅 수행을 위해 생성되는 테스트 하니스(Test Harness)[8]의 오버헤드를 극복하기 위하여 SCI 수준 제어를 포함한다. 이는 SCI의 수준을 다양하고 적절하게 선택하고 조절함으로써 소스코드에 대한 Instrumentation으로 발생하는 Code Size, Memory Size등의 오버헤드를 최소화하려는 노력이다. 또한, 대상 시스템의 특성과 문제를 다루기 위해 수행되어질 SCI의 종류(Performance, Coverage, Thread & Event Analyzer)를 선택한다.

· Code Analysis Component[4] : 동일 개발언어라 하여도 플랫폼에 따라 언어의 구성, 키워드 및 예약어가 다르기 때문에 각 플랫폼의 새로운 식별자를 해결할 방법이 필요하다. 이를 위해서 본 프레임워크에서는 기본적인 Parsing을 수행하고 해당 컴파일러에 특수한(타입, 인라인 코드식별) 식별자에 대해서는 컴파일러 단계로 넘기는 방법을 채택한다.

4. 결론

본 논문에서는 인스트루멘테이션에 동적-플러그인 기법을 사용하여 이질적인 임베디드 S/W개발환경에도 유연히 대응할 수 있는 임베디드 S/W 테스팅의 디자인 프레임 워크를 제시하였다.

그러나, 이러한 디자인 프레임워크는 알고리즘의 정교성, 설계의 복잡성, 구현의 난이도가 높다. 또한, 프로파일링 오버헤드를 최소화하고 수집되어지는 데이터의 정확성을 최대화하기 위해 좀더 구체적인 수집데이터 식별과 극복방안이 연구되어야 할 것이다.

참고문헌

[1] Bart Broekman and Edwin Notenboom, "Testing Embedded Software"

[2] 정보통신연구진흥원(IITA), "세계 임베디드 소프트웨어 시장분석", "소프트웨어 산업전망"

[3] Xingfu Wu, "Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System"

[4] Edu Metz, Raimondas Lencevicius, "Efficient Instrumentation For Efficient Instrumentation For Performance Profiling"

[5] Roxana Diaconescu & Michael Fran, "Performance Profiling System for a Dynamic Distributed Java Virtual Machine"

[6] William Y. Jhun, "Techniques for Estimating Worst-Case Program Execution Times with Unexpected Hardware Architecture Changes or Unknowns", In Proc Spring 2001

[7] Jakob Engblom, Andreas Ermedahl, Mikael Sjö din, Jan Gustafsson, Hans Hansson. "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems." In Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00), November 2000.

[8] Bart Broekman & Edwin Notenboom, "Testing Embedded Software", Addison-Wesley

[9] Rational Corporation, "Rational Unified Process"