임베디드 소프트웨어 테스트를 개선하기 위한 에뮬레이터 기반 인터페이스 테스트 도구

(An Interface Test Tool Based on an Emulator for Improving Embedded Software Testing)

서 주 영 † 최 병 주 ††
(Jooyoung Seo) (Byoungju Choi)

요 약 임베디드 시스템은 어플리케이션, OS 커널, 디바이스 드라이버, HAL, 하드웨어와 같은 이질적 계층들이 매우 밀접히 결합되어 있다. 임베디드 시스템은 제품 목적과 탑재된 하드웨어에 따라 맞춤 제작된다. 또한 점점 짧아지는 제품 주기 때문에 여러 업체의 소프트웨어, 하드웨어가 불안정한 상태에서 통합된다. 따라서 모든 계층에 결함 발생 확률이 높다. 임베디드 소프트웨어 개발자는 자신의 코드를 결함이 내재된 다른 계층들과 통합된 상태에서 테스트하며, 이 때문에 테스트해야 할 모든 영역을 테스트하였는지, 자신의 코드가 잘못된 건지, 통합된 다른 소프트웨어나 하드웨어에 문제가 있는 건 아닌지를 확신하기 힘들다. 본 논문은 임베디드 소프트웨어 개발자가 다양한 계층에 내재된 결함 위치와 원인을 추적할 수 있도록 하는 임베디드 소프트웨어 인터페이스 테스트 방안과 이를 구현한 자동화 도구 Justitia를 제안한다. 제안하는 기술은 개발자를 돕기위한 이뮬레이터를 이용한 디버깅을 전문적인 테스팅으로 승화시킨 자동화 방안이다.

키워드 : 임베디드 소프트웨어 테스팅, 테스트 도구, 임베디드 소프트웨어 인터페이스, 이뮬레이터

Abstract Embedded system is tightly coupled with heterogeneous layers such as application, OS kernel, device driver, HAL and hardware. Embedded system is customized for the specific purpose and hardware. In addition, the product cycle is so fast that software and hardware, which are developed by several vendors, are integrated together under unstable status. Therefore, there are lots of possibilities of faults in all layers. Because embedded software developers test their codes integrated with faulty layers, they cannot confirm 'whether testing of every aspects was completed, their code was failed, or integrated software/hardware has some problems'. In this paper, we propose an embedded software interface test method and a test tool called *Justitia* for detecting faults and tracing causes in the interface among heterogeneous layers. The proposed technique is an automated method which improves debugging upto professional testing using an emulator for helping developer.

Key words : Embedded software testing, Testing tool, Embedded software interface, Emulator

· 이 논문은 2006년 정부의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2006-531-D00027)

† 학생회원 : 이화여자대학교 컴퓨터정보통신학과
jyseo@ewhain.net
†† 종신회원 : 이화여자대학교 컴퓨터정보통신학과 교수
bjchoi@ewha.ac.kr

논문접수 : 2008년 1월 8일
심사완료 : 2008년 5월 22일

Copyright©2008 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 컴퓨팅의 실제 및 레터 제14권 제6호(2008.8)

1. 서 론

현재 임베디드 시스템의 10~20%를 임베디드 소프트웨어가 차지하나 시스템 결함의 80%이상이 하드웨어가 아닌 임베디드 소프트웨어로부터 야기되고 있다. 많은 임베디드 산업 현장에선 소프트웨어 품질이 심각하게 부각되면서 제품 품질 확보를 위한 사용성 테스트, 시스템 테스트, 인수 테스트와 같은 개발 프로세스 후반의 테스트는 전문 테스터에 의해 체계적으로 수행되고 있다. 그러나, 단위 테스트, 통합 테스트의 개발 프로세스 전반의 테스트는 아직까지도 내부 개발자의 책임 하에 묵시적으로 수행되고 있는 것이 현실이다[1,2]. 프로세스

후반 테스트는 테스트 대상 소프트웨어를 블랙박스로 간주한 테스트가 가능하기 때문에 테스트 기술을 전문적으로 훈련 받은 인력에 의한 체계화된 테스트가 가능하다. 그러나 프로세스 전반 테스트는 프로그램 단위를 한번에 하나씩 또는 합쳐서 수행하는 것으로 프로그램 내부 구조에 대한 상세한 지식을 요구하기 때문에 개발 담당자가 직접 수행할 수 밖에 없다. 상대적으로 테스트 기술이 미흡한 개발자의 테스팅은 발견된 결함을 디버깅하거나 시스템이 동작함을 증명하는 수준에 머무르고 있다. 결함 발견 시점이 개발 후반으로 갈수록 복구를 위한 비용은 기하급수적으로 증가하기 때문에, 결함 조기 발견과 밀접한 개발자의 테스트를 강화하는 것은 매우 중요한 일이다. 즉, 개발자에 의한 비전문적인 디버깅 활동을 전문적인 테스팅으로 강화시키는 테스트 방안이 개발자에게 제공되어야 한다. 테스트의 중요성을 알면서도 소홀한 이유는 테스트에 필요한 테스트 케이스, 테스트 드라이버와 테스트 스크립트를 준비하는 작업이 어렵고 귀찮기 때문이며, 특히 임베디드 소프트웨어 개발자가 겪는 테스트 어려움은 다음과 같은 임베디드 소프트웨어 특성 때문에 극대화된다.

첫째, 임베디드 시스템의 하드웨어와 소프트웨어들은 매우 밀접히 결합되어 있다[3]. 임베디드 소프트웨어는 일반 소프트웨어와 달리 제한된 하드웨어 자원 내에서 동작하기 때문에 효율적인 자원 공유를 위해 매우 최적화되어 있다. 이러한 특성은 임베디드 시스템을 구성하는 하드웨어와 소프트웨어를 따로 분리하여 독립 적으로 실행하거나 테스트하기 매우 까다롭게 만든다. 이론적으론 점진적 방식으로 통합되어 상호작용하는 두 구성요소를 단계적으로 테스트하여야 하나, 임베디드 소프트웨어는 시스템 계층 간 높은 결합력 때문에 빅뱅(big-bang) 방식으로 전체가 통합된 상태에서 테스트를 수행하는 것이 일반적인 현황이다. 개발자는 자신의 코드 이외에 구체적 행위를 알 수 없는 외부 코드, 특히 하드웨어와 통합하여 테스트를 수행할 수 밖에 없다.

둘째, 임베디드 제품의 하드웨어와 소프트웨어들은 매우 다양한 개발 업체들이 참여하여 구현된다. 일반 소프트웨어의 경우, 상대적으로 매우 안정화된 하드웨어 플랫폼에서 운영되고 함께 동작하는 다른 소프트웨어 와도 매우 독립적이다. 그러나, 임베디드 소프트웨어는 제품 목적과 탑재된 하드웨어에 따라 매번 맞춤제작되고, 점점 짧아지는 제품 주기 때문에 여러 업체의 안정화되지 않은 하드웨어와 소프트웨어가 동시에 자주 통합되고 테스트된다. 이 때문에 개발자가 테스트하려는 자신의 코드 이외의 통합된 다른 계층에도 결함 발생 가능성은 고르게 분포하게 된다.

셋째, 개발자는 자신이 구현한 코드의 구조는 잘 알고 있지만 통합되는 다른 개발자, 다른 업체에 의해 구현된 하드웨어, 소프트웨어의 내부는 블랙박스로서 알기 힘들다. 동시에 통합되어 테스트될 수 밖에 없는 임베디드 소프트웨어 특성 상 블랙박스 영역이 너무 광범위하기 때문에 결함을 발견하더라도 결함 발생 위치와 결함 원인을 추적하기 힘들다. 즉, 개발자는 항상 테스트해야 할 모든 영역을 테스트 하였는지, 자신의 코드가 잘못된 건지, 아니면 통합된 다른 소프트웨어가 잘못된 건지, 혹시 하드웨어에 문제가 있는 건 아닌지를 배포하는 순간 까지도 확신할 수 없다.

이러한 임베디드 소프트웨어 테스트 환경에서 모든 계층에 분포되어 있을 수 있는 결함 위치와 원인을 추적하기 위해선, 임베디드 시스템을 구성하는 이질적 계층의 하드웨어, 소프트웨어들 사이의 상호작용인 인터페이스가 반드시 테스트되어야 한다. 즉, 이질적 계층의 인터페이스는 일반 소프트웨어와 차별화되는 임베디드 소프트웨어의 핵심 테스트 영역임과 동시에 결함 발견 및 임베디드 시스템 계층 간의 결함 원인 유추를 위한 모니터링 대상이 되어야 한다.

본 논문에서는 이러한 임베디드 소프트웨어 테스트의 어려움을 해결하고, 개발자에 의해 수행되는 테스트를 강화하기 위한 임베디드 소프트웨어 인터페이스 테스트 방안과 자동화 도구인 Justitia를 제안한다.

제안하는 테스트 기술은 컴파일러가 생성하는 디버깅 정보와 개발자가 디버깅 목적으로 사용하는 이뮬레이터를 활용하여, 어떻게 인터페이스를 추출할 것인가, 무엇이 인터페이스에서 테스트되어야 하는가, 어떻게 인터페이스에서 결함 판정을 할 것인가의 핵심 테스트 활동을 해결하였다. 또한, 구현한 Justitia를 검증하기 위해 임베디드 리눅스 기반 스마트폰에 탑재된 임베디드 소프트웨어를 대상으로 인터페이스 비중, 인터페이스 테스트 커버리지와 인터페이스 결함을 분석하는 사례 연구를 수행하였다.

본 논문의 구성은 다음과 같다. 2장과 3장은 임베디드 소프트웨어 인터페이스 테스트 방안과 자동화 도구 Justitia를 기술한다. 4장은 Justitia를 적용한 사례 연구를, 5장은 기존 임베디드 소프트웨어 테스트 도구를 비교 분석하며, 마지막으로 6장에서 결론을 맺는다.

2. 임베디드 소프트웨어 인터페이스 테스트 방안

본 논문은 임베디드 시스템 내의 특정 소프트웨어와 연관된 인터페이스를 테스트하기 위해 다음의 테스트 이슈를 해결하는 인터페이스 정보 추출 기법, 테스트 케이스 선정 기법, 테스트 수행 기법을 제안한다.

(1) 어떻게 인터페이스를 추출할 것인가

개발자는 자신의 코드는 잘 알지만 자신의 코드와 연관되어 테스트되어야 할 위치는 너무 광범위하다. 특히 임베디드의 경우, 운영체제나 하드웨어와 같은 여러 분야의 지식이 필요하기 때문에 인터페이스를 식별하기란 매우 힘들다. 제안하는 테스트 방안은 디버깅 정보를 통해 임베디드 시스템 전체 구조를 자동으로 파악하고, 개발자가 테스트하려는 소프트웨어와 물리적으로, 논리적으로 연관된 인터페이스를 자동 추출하는 인터페이스 정보 추출 기법을 포함한다.

(2) 무엇이 인터페이스에서 테스트되어야 하는가

제안하는 테스트 방안은 임베디드 시스템 계층의 상호 작용 유형을 분석하여 인터페이스 유형을 정의하고, 인터페이스 유형 별 테스트되어야 할 항목을 결정하는 인터페이스 테스트 케이스 선정 기법을 포함한다.

(3) 어떻게 인터페이스에서 결함 판정을 할 것인가

제안하는 테스트 방안은 이뮬레이터의 디버깅 기능을 활용하여 테스트 수행을 자동화하고, 결함 판정에 필요한 입력 데이타와 예상 출력을 모니터링할 수 있도록 하는 테스트 수행 기법을 포함한다.

2.1 인터페이스 정보 추출 기법

인터페이스 정보 추출 기법은 컴파일러가 생성하는 임베디드 소프트웨어의 디버깅 정보를 분석하여 인터페이스 테스트 케이스 선정을 위한 정보를 추출하는 알고리즘이 핵심이다.

알고리즘: 인터페이스 정보 추출
입력: 임베디드 소프트웨어 디버깅 정보
출력: 임베디드 소프트웨어 인터페이스 정보

단계1: 임베디드 소프트웨어 계층 분석
단계1.1: 계층 추출
단계2: 임베디드 소프트웨어 유닛 분석
단계2.1: 함수 호출 관계 추출
단계2.2: 하드웨어 제어 명령문 추출
단계2.3: 유닛 분류(HUd, HUi, SUk, SUd, SUp)

(1) 임베디드 시스템 계층 및 유닛

임베디드 시스템은 일반적으로 그림 1과 같이 어플리케이션, 운영체제, 디바이스 드라이버, HAL(Hardware Abstraction Layer), 하드웨어와 같은 이질적 계층으로 구성된다[3,4]. 본 논문은 특히 그림1의 운영체제 계층과 하드웨어 계층 사이의 인터페이스에 초점을 두고 있으며, 이들 계층들을 SUk, SUd, SUp, HUd, HUi의 5개 유닛으로 분류하고, 이들 사이의 상호 작용을 임베디드 소프트웨어 테스트 시 반드시 확인해야 할 핵심 테스트 위치인 인터페이스로 식별한다.

- SUk는 커널 기능의 시스템 콜 함수나 API함수로 구현된 운영체제 서비스 유닛을 의미한다.

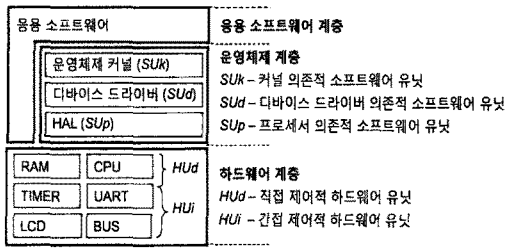

그림 1 임베디드 시스템의 계층과 유닛

- SUd는 LCD, USB, UART와 같은 디바이스를 제어하는 기능을 구현한 소프트웨어 유닛을 의미한다.
- SUp는 하드웨어를 제어하는 HAL 계층의 타겟 프로세서 의존적인 소프트웨어 유닛을 의미한다.
- HUd는 소프트웨어 유닛이 직접 통신하는 메모리와 레지스터와 같은 하드웨어 유닛을 의미한다.
- HUi는 LCD, USB, UART와 같은 디바이스로서 HUd에 의해 간접 제어되는 하드웨어 유닛을 의미한다.

(2) 호출 관계

인터페이스는 서로 다른 5개 유형의 유닛들이 상호 작용하는 위치이며, 호출 관계는 소프트웨어 유닛(SUk, SUd, SUp)과의 인터페이스의 경우 해당 유닛의 함수 호출 문장을 의미하고, 하드웨어 유닛(HUd, HUi)과의 인터페이스는 해당 하드웨어 유닛을 제어하는 명령문을 의미한다. 인터페이스의 호출 관계는 테스트 대상의 디버깅 정보 중 역어셈블 된 코드의 BRANCH와 같은 특정 기계명령어를 이용하여 간단히 추출할 수 있다.

2.2 인터페이스 테스트 케이스 선정 기법

테스트 케이스 선정 기법은 테스트 대상 소프트웨어의 디버깅 정보로부터 추출된 임베디드 소프트웨어 인터페이스 정보를 바탕으로 인터페이스를 테스트하기 위한 테스트 케이스를 선정하는 알고리즘이 핵심이다. 인터페이스 테스트 케이스는 아래와 같이 인터페이스 위치, 인터페이스 테스트 항목, 인터페이스 심볼, 입력 데이타와 예상 출력으로 구성된다.

인터페이스 테스트 케이스 =
{인터페이스 위치, 인터페이스 항목,
인터페이스 심볼, 입력과 예상 출력}
{인터페이스 위치} 서로 다른 하드웨어, 소프트웨어 계층에 속하는 유닛 사이의 상호작용하는 코드 위치
{인터페이스 항목} 인터페이스 위치에서의 테스트 항목
{인터페이스 심볼} 인터페이스 항목에 따라 결함 판정에 영향을 미치는 변수
{입력과 예상 출력} 인터페이스 심볼의 입력 값과 예상되는 출력 값

알고리즘: 인터페이스 테스트 케이스 선정
입력: EmITM(Embedded System Interface Test Model),

임베디드 소프트웨어 인터페이스 정보
출력: 인터페이스 테스트 케이스

단계1: 임베디드 소프트웨어 인터페이스 식별
단계1.1: 인터페이스 유형에 매칭되는 인터페이스 추출 (HPI_1, HPI_2, OPI_1, OPI_2, OPI_3)
단계2: 인터페이스 테스트 케이스 생성
단계2.1: 추출한 인터페이스 유형에 대응되는 EmITM의 인터페이스 항목 식별
단계2.2: 대응된 인터페이스 항목에 따라 인터페이스 심볼, 입력 데이타와 예상 출력을 식별

(1) 임베디드 소프트웨어 인터페이스 유형

임베디드 소프트웨어 인터페이스에서 테스트되어야 할 인터페이스 항목은 임베디드 시스템 인터페이스 테스트 모델(EmITM: Embedded System Interface Test Model)에 관한 연구[5]를 통해 이미 정의하였다. 본 논문 에서는 테스트 항목에 대응되는 인터페이스를 자동 추출 하기 위해 앞서 정의한 5개 유닛의 상호작용 유형을 분석하여 그림 2와 같은 5개의 인터페이스 유형을 정의 하였다. 즉, 인터페이스 유형 별로 테스트되어야 할 인터페이스 항목이 자동으로 결정되며, 각 항목 별로 결함 판정에 관련된 인터페이스 심볼, 입력과 예상 출력도 결정된다.

- HPI_1(Hardware Part Interface)은 SUd, SUp 소프트웨어 유닛에서 메모리와 같은 읽기/쓰기 가능한 하드웨어 유닛(HUd)에 직접 접근하여 값을 읽거나 쓰는 인터페이스이다.
- HPI_2는 SUd 또는 SUp 소프트웨어 유닛에서 레지스터(HUd)의 주소를 통해 타이머, LED, UART와 같은 다른 하드웨어 유닛(HUi)을 간접 제어하는 인터페이스이다.
- OPI_1(Operating System Part Interface)는 SUd와 SUp에서 하드웨어와 직접 관련이 없는 SUk의 함수를 호출하는 인터페이스이다. 이때, 호출되는 SUk의 함수에는 태스크 관리, 태스크 통신, 예외 처리와 관련된 시스템 콜 함수 및 API함수가 해당된다.
- OPI_2는 SUd와 SUp에서 하드웨어와 간접 관련이 있는 SUk의 함수를 호출하는 인터페이스이다. 이때, 호출되는 SUk의 함수에는 가상주소관리를 위한 시스템 콜 함수 및 API함수가 해당된다.
- OPI_3는 SUd와 SUp에서 하드웨어와 직접 관련 있는 SUk의 함수를 호출하는 인터페이스이다. 이때, 호출되는 SUk의 함수에는 물리적인 메모리 및 타이머와 같이 실제 하드웨어(HUd) 제어를 위한, 메모리 관리, 시간 관리, 인터럽트 처리, IO관리, 네트워킹, 파일 시스템을 위한 시스템 콜 함수 및 API함수가 해당된다.

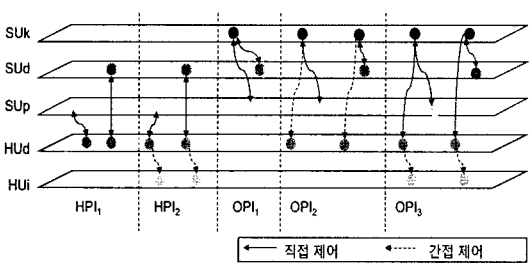

그림 2 임베디드 소프트웨어 인터페이스 유형

(2) EmITM(Embedded System Interface Test Model)

EmITM은 하드웨어 인터페이스(HPI: Hardware part interface) 및 운영체제 인터페이스(OSI: OS part interface)에 대한 테스트 항목을 정의한 모델이다. EmITM의 HPI는 임베디드 시스템을 구성하는 하드웨어 설계 문서를 토대로 추출한 것으로 메모리, 입출력 디바이스, 타이머에 대하여 총 15개의 인터페이스 항목으로 정의한다. OPI는 POSIX1003.4, ELCPS, KELPS을 근거로 태스크 관리, 태스크 간 통신, 타이머 관리, 예외 및 인터럽트 처리, 메모리 관리, 입출력 관리, 네트워킹, 파일 시스템에 관한 총 106개의 인터페이스 항목으로 정의되어 있다. 표 1은 EmITM을 요약한 것이다. 우리는 앞서 추출한 인터페이스 유형에 따라 EmITM에 정의된 인터페이스 항목을 대응시킴으로 테스트해야 할 항목을 자동으로 결정한다. 테스트 케이스 중 인터페이스 심볼은 인터페이스 위치에서 결함 판정에 연관되는 소프트웨어/하드웨어 변수를 의미하며, 입력 데이타와 예상 출력을 모니터링하는 역할을 한다. 인터페이스 심볼, 입력 데이타와 예상 출력은 본 테스트 방안을 적용하는 테스트 대상에 따라 EmITM을 맞춤 정의하여 결정되며, 본 방안을 통해 구현한 테스트 도구인 Justitia의 경우 리눅스 기반 임베디드 소프트웨어를 위한 인터페이스 모델에 관한 연구[6]를 통해 EmITM의 인터페이스 항목을 리눅스 도메인에 맞추어 인터페이스 심볼, 입력과 예상 출력을 정의하였다. 이에 관한 구체적인 테스트 케이스 선정의 예는 2.4절 예제를 통해 기술한다.

2.3 테스트 수행 기법

테스트 수행 기법은 선정된 테스트 케이스의 수행과 결과 분석을 자동화하기 위한 방안이다. 일반적으로 임베디드 소프트웨어 테스트 시 타겟 보드를 제어하고 테스트 결과를 모니터링 하기 위해 별도의 테스트 에이전트(test agent)를 구현하거나, 원본 소스 코드에 테스트 코드를 삽입하는 방법을 사용한다. 이 두 가지 방법은 임베디드 소프트웨어 테스트 시 다음과 같은 문제점이 존재한다.

테스트 에이전트의 경우, 테스트 대상 소프트웨어와 동일한 타겟 보드에 탑재되어 모니터링하는 방법이기 때문에, 테스트하려는 임베디드 소프트웨어와 자원을 공

표 1 EmITM(Embedded System Interface Test Model)[5]

인터페이스		계층	유닛 HUd (HPI의 경우) / SUk (OPI의 경우)	인터페이스 항목(세부 인터페이스 항목 수)	
HPI	HPI_1	- SUd, HUd - SUp, HUd	RAM	HP1 메모리	- 메모리 할당 (1) - 할당 받은 주소에 대한 접근 (2)
	HPI_2	- SUd, HUi - SUp, HUi	I/O peripheral registers	HP2 입출력 디바이스	- 입출력 연산 (7)
			Timer peripheral registers	HP3 타이머	- 타이머 컨트롤 레지스터 설정 (2) - 타이밍 파라미터 설정 (2) - 작업 처리 시간 (1)
OPI	OPI_1	- SUd, SUk - SUp, SUk	_JOB_CONTROL, _SINGLE_PROCESS, _SPAWN, _SCHEDULING, _PRIORITY_SCHEDULING, _MULTI_PROCESS, _THREADS_EXT, _MULTI_ADDR_SPACE, _THREADS _THREAD_PROCESS_SHARED, _THREADS_REALTIME, _THREAD_PRIORITY_SCHEDULING, _THEADS_REALTIME_EXT, _SPIN_LOCKS, _BARRIORES	OP1 태스크 관리	- 프로세스 생성, 전환, 종료 (4) - 멀티 스레드 어플리케이션 및 멀티 프로세싱 지원 (4) - 스케줄링 (6)
			_IPC, _SEMAPHORES, _PIPE	OP2 태스크 간 통신	- 동기화 (7) - 파이프, 세마포어와 같은 프로세스간 통신 지원 (18)
			_SIGNALS, _SIGNAL_JUMP, _REALTIME_SIGNALS	OP5 예외 처리	- 정상 모드로의 복귀 (2) - 예외 처리 (3)
	OPI_2	- SUd, SUk - SUp, SUk	_DYNAMIC_LINKING, _MEM_MGMT, _MAPPED_FILES, _MEM_LOCK, _MEMORY_PROTECTION, _MEM_LOCK_RANGE, _MULTI_ADDR_SPACE, _REG_EXP	OP6 가상 메모리 관리	- 메모리 할당 (1) - 할당 받은 주소에 대한 접근 (1) - 올바른 메모리 해제 (3) - 페이지 폴트 (1) - 임계 영역 보호 (2)
	OPI_3	- SUd, SUk, HUd - SUp, SUk, HUd	_DYNAMIC_LINKING, _MEM_MGMT _MAPPED_FILES, _MEM_LOCK, _MEMORY_PROTECTION, _MEM_LOCK_RANGE, _MULTI_ADDR_SPACE, _REG_EXP	OP6 물리 메모리 관리	- 메모리 할당 (1) - 할당 받은 주소에 대한 접근 (1) - 올바른 메모리 해제 (3) - 페이지 폴트 (1) - 임계 영역 보호 (2)
			_TIMERS, _CLOCK_SELECTION	OP3 타이머 관리	- 시간 지연 (1) - 시간 값 설정 및 읽기 (6)
			_SIGNALS, _SIGNAL_JUMP, _REALTIME_SIGNALS	OP4 인터럽트 처리	- 정상 모드로의 복귀 (2) - 인터럽트 처리 (15)
			_ASYNCHRONOUS_IO, _DEVICE_IO, _DEMULTI_ADDR_SPACE, _WIDE_CHAR_DEVICE_IO, _DEVICE_SPECIFIC, _DEVICE_SPECIFIC_R	OP7 입출력 관리	- 디바이스 드라이버 특성 지원 (1) - 디바이스 I/O request 처리 (6)
			_NETWORKING, _NETWORKING_RPC	OP8 네트워킹	- 올바른 네트워킹 자료 구조 (2) - 해당 네트워크를 지원하는 시스템 콜의 올바른 사용 (3) - 패킷 전송/받기 (5)
			_FD_MGMT, _FIFO, _FILE_SYSTEM, _SYNCHRONOZIED_IO, _FYNC, _FILE_ATTRIBUTES, _FILE_SYSTEM_EXTEN, _FILE_SYSTEM_R, _LARGE_FILE, _STDIO_LOCKING	OP9 파일 시스템	- 디스크 공간 관리 (1) - 올바른 파일 읽기와 쓰기 (3)

유해야 하고 모니터링할 수 있는 내용도 제한적이다.

테스트 코드 삽입의 경우, 개발자들도 디버깅을 위해 흔히 사용하는 방법으로 모니터링 하기 원하는 변수의 값을 확인할 수는 있지만, 메모리 제약과 타이밍 이슈가 심각한 임베디드 시스템 특성 상 변경된 코드로 인해 시스템이 오동작할 수 있는 문제점이 있다.

본 논문은 이러한 문제점을 지양하기 위해 일반적으로 임베디드 소프트웨어 개발자가 디버깅을 위해 사용하는 이뮬레이터를 타겟 보드 제어와 테스트 결과 모니터링에 활용하는 테스트 수행 방안을 제안한다. 즉, 인터페이스 위치에 중단점(breakpoint)을 설정하고, 인터페이스 심볼을 모니터링하여 그 값이 예상 출력과 같은지를 비교함으로써 결함 판정을 자동화하는 것이 테스트 스크립트 생성 알고리즘의 핵심이다. 테스트 스크립트는 go/step, break, monitoring, if/else, print의 명령으로 구성된다.

테스트 스크립트 = {go/step, break, monitoring, if/else, print}
{go/step} 인터페이스 항목에 따라 타겟 보드를 제어하기 위한 CPU실행 명령
{break} 인터페이스 위치를 테스트하기 위한 중단점 설정 명령
{monitoring} 인터페이스 심볼의 현재 값을 측정하기 위한 레지스터, 변수, 메모리의 모니터링 명령
{if/else} 결함 판정을 위해 모니터링 값과 예상 출력을 비교하는 명령
{print} 결함 판정 결과를 로그로 저장하는 명령
알고리즘: 테스트 스크립트 생성
입력: 인터페이스 테스트 케이스
출력: 테스트 스크립트

단계1: 테스트 실행 스크립트 생성

단계1.1: break 명령을 이용하여 인터페이스 위치에 중단점 설정
단계1.2: go/step 명령을 이용하여 타겟 보드 실행
단계2: 결함 판정 스크립트 생성
단계2.1: monitoring 명령을 이용하여 인터페이스 심볼의 현재 값 측정
단계2.2: if/else 명령을 이용하여 예상 출력과 현재 값을 비교하여 결함 판정
단계3: 테스트 결과 저장 스크립트 생성
단계3.1: print 명령을 이용하여 테스트 로그를 저장

2.4 알고리즘 적용 예

다음은 ARM9T기반 임베디드 리눅스 LCD 디바이스 드라이버를 대상으로 인터페이스 정보 추출, 테스트 케이스 선정, 테스트 스크립트 생성 알고리즘을 설명한 예이며, 그림 3은 이를 도식화한 것이다.

(1) 인터페이스 정보 추출

테스트 대상인 LCD 디바이스 드라이버는 그림 3(a)와 같이 커널, 비디오, HAL, 하드웨어 계층과 상호작용하는 임베디드 소프트웨어이다. 그림 3(a)에서 보듯이 커널 계층에 속하는 박스들은 SUk이고, 비디오 계층은 SUd, HAL은 프로세서를 제어하는 SUp, 하드웨어 계층은 HUd와 HUi로 분류할 수 있다. 그림 3(a)는 이들 서로 다른 계층 사이의 상호작용을 호출 그래프 형식으로 도식화한 것이며, 하드웨어와의 상호작용의 경우 하드웨어 제어 명령문을 하나의 호출로 간주하여 작성하였다. 즉, 호출 그래프의 회색박스가 테스트 되어야 할 이질적 계층 사이의 인터페이스를 갖는 유닛을 의미한다.

(2) 테스트 케이스 선정

그림 3(b)는 그림 3(a)의 네모로 표시된 부분을 확대하여 나타낸 것으로서 HAL 계층의 fb_init_fbinfo() 함수가 LCD 프레임버퍼 초기화를 위해 커널 계층의 kmalloc() 함수를 호출하는 인터페이스의 예이다. 즉,

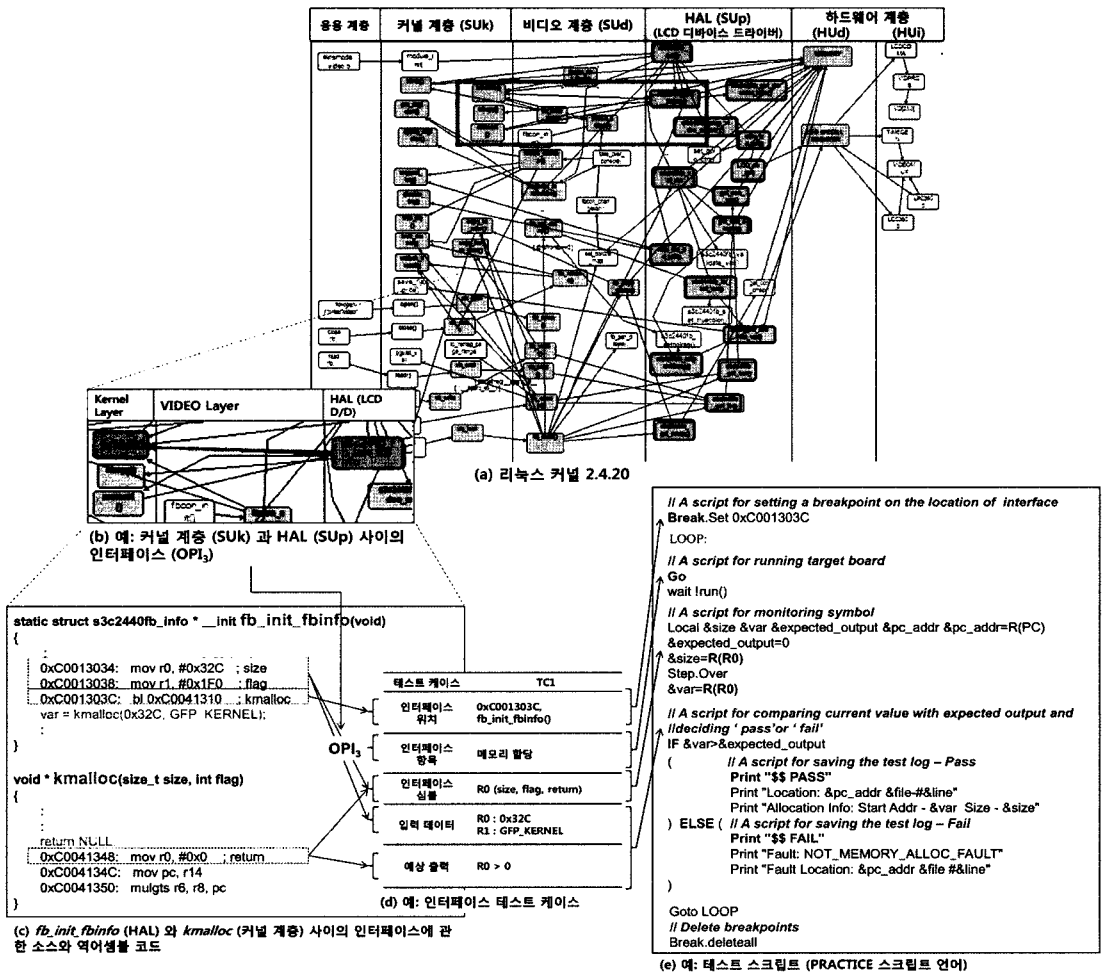

그림 3 임베디드 리눅스 LCD 디바이스 드라이버의 예

SUp와 SUk 사이의 인터페이스로 우리가 정의한 인터페이스 OPI_3에 해당한다. 인터페이스 위치는 kmalloc() 호출 위치인 0xC001303C가 되며, 예제와 같은 ARM기반 프로세서의 경우 BL명령으로 간단히 추출할 수 있다. 또한, SUk가 메모리 할당 함수이기 때문에 표 1의 EmITM에 의해 메모리 할당 인터페이스 항목이 테스트되어야 한다.

리눅스 기반 임베디드 소프트웨어를 위한 인터페이스 모델에 관한 연구[6]를 통해 EmITM의 인터페이스 항목을 리눅스 도메인에 맞추어 인터페이스 심볼, 입력과 예상 출력을 정의하였다. 메모리 할당 예의 경우, 그림 3(c)와 같이 kmalloc() 함수 호출을 위한 size, flag 인자의 값이 입력으로, kmalloc() 함수 호출 결과인 리턴 값이 예상출력으로 모니터링되어야 한다.

함수의 인자나 반환 값은 ARM기반 프로세서 명세의 표준 함수 호출 규약[7]에 따라 R0, R1, R2, R3의 레지스터로 제한되기 때문에, 다양한 변수를 모니터링 하는 대신 제한된 레지스터를 인터페이스 심볼로 모니터링할 수 있다. 이 예의 경우, 그림 3(d)와 같은 인터페이스 테스트 케이스가 선정된다.

(3) 테스트 스크립트 생성

그림 3(d)의 테스트 케이스를 실행하기 위해선, fb_init_fbinfo() 함수가 kmalloc() 함수를 호출하기 직전까지 실행하고(BREAK RUN), kmalloc() 함수의 입력이 유효한 값인지 확인한 후, kmalloc() 함수를 실행 시키고(STEP OVER), 실행 직후 반환 값을 모니터링 하여 예상 출력과 비교함으로 결함을 판정한다. 이 과정을 위한 스크립트는 그림 3(e)이다. 인터페이스 위치에 break 명령을, 인터페이스 항목에 따라 go/step 명령을, 결함 판정을 위한 인터페이스 심볼을 모니터링 하도록 register명령을 사용하는 형태로 작성된다.

3. 구현 및 설계

우리는 앞서 기술한 임베디드 소프트웨어 인터페이스 테스트 방안에 기반하여 테스트케이스 선정, 테스트 드라이버 생성, 테스트 자동 수행과 테스트 커버리지 분석의 핵심 테스트 활동을 자동화한 임베디드 소프트웨어 인터페이스 테스트 도구인 Justitia를 구현 하였다[6]. Justitia는 Windows XP 운영체제 하에서 C/C++로 개발되었고, ARM9T계열의 32bit 마이크로 프로세서의 임베디드 리눅스 2.4.x, 2.6.x 버전과 ARM11계열의 64bit 마이크로프로세서의 임베디드 리눅스 2.6.x버전 임베디드 소프트웨어를 위한 테스트 도구로 활용할 수 있다.

Justitia는 gcc컴파일러가 생성하는 ELF(Executable and Linking Format)의 디버깅 정보를 이용하며 JTAG 버전 TRACE32-ICD 이뮬레이터를 채택하였다. Justitia는 개발자가 이뮬레이터를 이용하여 일련의 디버깅을 하는 대신 임베디드 소프트웨어를 위한 핵심 테스트 위치인 인터페이스를 중심으로 테스트 활동을 강화할 수 있도록 한 자동화 도구로서 그림 4와 같이 테스트 대상 분석기, 테스트 케이스 마법사, 테스트 드라이버 마법사, 테스트 제어기, 테스트 결과 분석기, 보고서 생성기의 6개 핵심 모듈로 구성된다.

(1) 테스트 대상 분석기

테스트 대상 분석기는 사용자로부터 테스트 대상 임베디드 소프트웨어의 오브젝트코드를 입력 받아 디버깅 정보를 추출하여, 이로부터 EmITM 기반의 인터페이스 위치 파악을 위한 소프트웨어와 하드웨어 계층 구조와 호출 관계의 인터페이스 정보를 분석하고 호출 관계 그래프를 생성한다(그림 5(a)).

(2) 테스트 케이스 마법사

테스트 케이스 마법사는 인터페이스 정보와 호출 관계 그래프를 입력으로 인터페이스 테스트 케이스를 자동 생성한다. 즉, 서로 다른 이질적 계층 간의 인터페이스를 테스트 위치로 식별하고, 각각의 인터페이스 위치에서 테스트되어야 할 항목을 대응시켜 테스트 케이스로 선정한다(그림 5(b)).

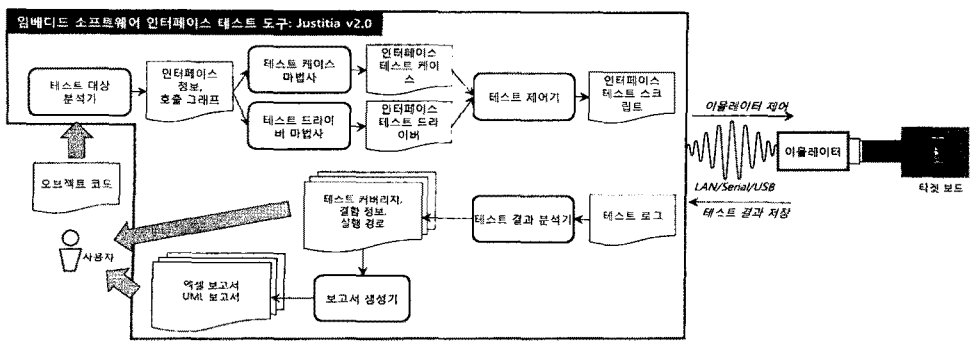

그림 4 임베디드 소프트웨어 인터페이스 테스트 도구Justitia의 데이타 흐름 그래프

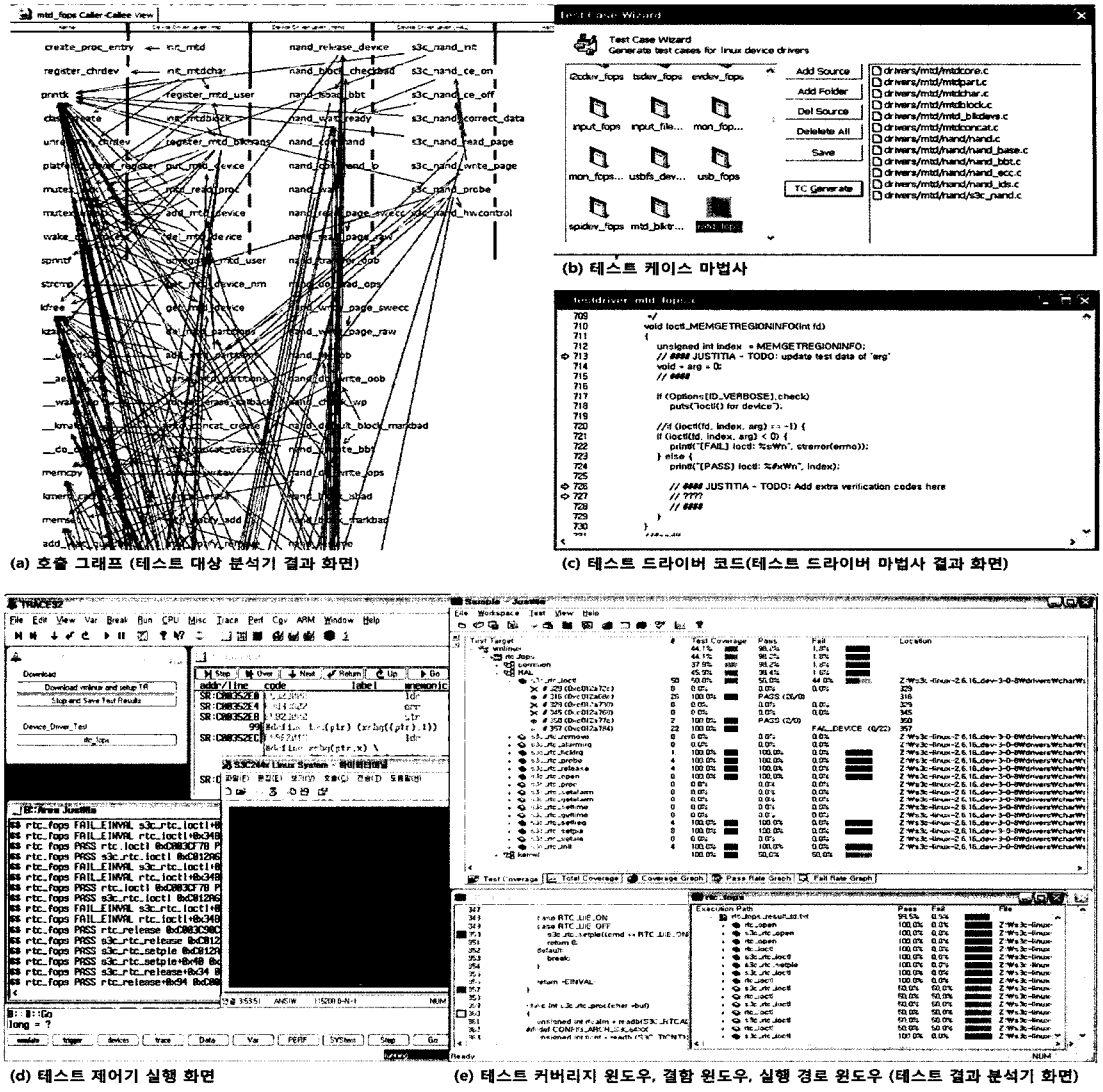

그림 5 Justitia 실행 화면

(3) 테스트 드라이버 마법사

테스트 드라이버 마법사는 인터페이스 정보를 입력으로 테스트 대상 임베디드 소프트웨어의 인터페이스를 한 번씩 호출하는 기능의 테스트 드라이버 소스 코드를 생성한다(그림 5(c)).

(4) 테스트 제어기

테스트 제어기는 이뮬레이터를 통해 타겟 보드를 제어하고, 테스트 케이스를 자동 실행시키며, 테스트 결과를 로그 파일로 저장하는 테스트 스크립트를 생성하며, 이를 배치 실행시킴으로 테스트 수행을 자동화한다(그림 5(d)).

(5) 테스트 결과 분석기

테스트 결과 분석기는 이뮬레이터에 의해 저장된 테스트 로그로부터 테스트 커버리지, 결함 정보, 실행 경로의 테스트 결과를 분석한다. 테스트가 어느 정도 진행되었는지를 나타내는 커버리지와 함께 테스트된 인터페이스 영역과 테스트되지 않은 영역을 구분함으로 파악할 수 있도록 한다. 이뮬레이터에서 단순히 제공하는 디버깅 기능을 테스트 커버리지와 관련한 테스팅에 통합함으로 테스트 커버리지 정보 이외에도 결함 발생 여부와 발생 위치, 발생 원인과 같은 결함 정보를 함께 분석한다(그림 5(e)).

(6) 보고서 생성기

보고서 생성기는 테스트 대상 분석 정보를 UML설계 문서로 가시화하고 테스트 결과를 엑셀 파일 및 UML 파일로 생성한다.

4. 사례 연구

본 절에서는 인터페이스가 임베디드 소프트웨어 테스트를 위한 유용한 위치인지와 임베디드 소프트웨어 인터페이스 테스트 도구 Justitia가 하드웨어, HAL, 디바이스 드라이버, 운영체제 커널의 다양한 이질적 계층 사이의 인터페이스에서 발생하는 결함을 발견하는 데 유용한지를 사례 연구를 통해 보인다. 이를 위해 Justitia를 스마트폰을 위한 임베디드 리눅스 기반 ARM계열 프로세서를 위한 임베디드 소프트웨어 테스트에 적용하였다.

4.1 테스트 대상

임베디드 리눅스 v2.6.16의 ARM9T기반 마이크로 프로세서 *A*와 ARM11 기반 마이크로 프로세서 *B*의 2개 버전 BSP의 6개 디바이스 드라이버를 테스트 대상으로 사례 연구하였다. 표 2의 S6 디바이스 드라이버의 경우는 *A*와 *B* 마이크로 프로세서에 동일한 코드가 사용되기 때문에 *A*의 한 개 버전만 테스트하였다.

표 2 테스트 대상 임베디드 소프트웨어

테스트 대상	*A* 마이크로 프로세서			*B* 마이크로 프로세서		
	LOC	호출	인터페이스	LOC	호출	인터페이스
S1	4586	435	192	7768	674	368
S2	827	114	70	1127	87	75
S3	495	76	41	896	76	35
S4	6001	698	220	9324	1024	229
S5	1013	210	132	2861	63	71
S6	3265	416	110	-	-	-

4.2 분석 및 결과

Justitia를 통해 각 테스트 대상에 대해 인터페이스가 전체 호출에서 차지하는 비중을 분석하고, 테스트 결과로 인터페이스 테스트 커버리지와 인터페이스 결함을 분석하였다.

(1) 인터페이스 비중

인터페이스 비중은 전체 호출 수(하드웨어와의 상호작용도 한 개의 호출로 계산) 중 이질적 계층 사이의 인터페이스가 차지하는 비중을 백분율로 계산한 것으로 임베디드 소프트웨어가 인터페이스를 중심으로 밀접히 결합되어 있다는 기본 특징을 확인할 수 있는 메트릭이다. 그 결과 표 3과 그림 6(a)와 같이 전체 호출 중, 표 2의 *A* 마이크로 프로세서는 46.7%, *B*는 60.0%, 총 평균 53.4%가 서로 다른 업체에 의해 개발되어 내부 구조를 알기 힘든 이질적 계층 사이의 인터페이스이다. 일반 소프트웨어에 비해 통합되는 모든 계층에서 결함 발생 확률이 높기 때문에, 분석 결과와 같이 인터페이스를 중심으로 한 높은 결합력은 인터페이스에서의 결함 발생 확률을 가중시킨다. 즉, 인터페이스는 임베디드 소프트웨어의 결함 발견과 결함 원인 식별을 위한 핵심 위치이며, 이를 커버하는 테스트가 매우 중요하다.

(2) 인터페이스 테스트 커버리지

인터페이스 테스트 커버리지는 전체 인터페이스 중 테스트된 인터페이스 수를 백분율로 계산한다. 테스트 커버리지는 결함 발견률과 밀접한 관계가 있으며, 테스트로 커버되는 영역이 많아질수록 결함 발견율도 높아지는 것으로 알려져 있다. 표 2의 전체 1839개의 인터페이스를 최소 한 번씩 테스트되도록 하기 위해 Justitia가 자동 생성해주는 테스트 드라이버 코드를 이용하여, *A* 마이크로 프로세서는 90.1%, *B*는 80.5%의 전체 85.3% 커버리지를 갖는 테스트를 수행하였다(표 3, 그림 6(b)). 테스트되지 않은 전체 14.7%의 인터페이스를 분석한 결과, 6.9%는 컴파일 조건에 의해 선택적으로 생성되는 코드이고, 7.8%는 사용하지 않는 코드(dead code)이었다. 즉, Justitia가 자동 생성하는 테스트 드라이버 코드를 활용하여 컴파일 조건과 사용하지 않는 코드를 제외한 인터페이스에 대해선 최소 한 번씩 실행되는 테스트를 수행할 수 있다.

(3) 인터페이스 결함

인터페이스 결함은 서로 이질적 계층이 통합되는 과정에서 발생하는 결함을 의미하며, Justitia를 표 2의 테스트 대상에 적용하여 전체 1839개의 인터페이스에서 *A* 마이크로 프로세서는 43개, *B*는 50개의 총 93개 인터페이스 결함을 발견하였다(표 3, 그림 6(c)).

발견한 인터페이스 결함의 발생 원인을 분석한 결과 단순 결함이 아닌 메모리 할당 및 충돌 문제, 인터럽트 처리 문제, 하드웨어 문제와 같이 이를 디버깅하기 위해선 운영체제나 하드웨어에 관한 지식과 경험이 상당히 필요로 되는 결함들이었다. 즉, Justitia를 통해 개인차가 심한 테스트를 자동화함으로써 위험도가 높은 인터

표 3 Justitia 테스트 결과

테스트 대상	*A* 마이크로 프로세서			*B* 마이크로 프로세서			TOTAL		
	비중	커버리지	결함	비중	커버리지	결함	비중	커버리지	결함
S1	44.1%	82.2%	2	54.6%	70.5%	15	49.4%	76.4%	17
S2	61.4%	90.2%	0	86.2%	90.2%	1	73.8%	90.2%	1
S3	53.9%	96.1%	1	46.1%	85.7%	1	50.0%	90.9%	2
S4	31.5%	93.3%	14	54.6%	78.2%	14	43.1%	85.8%	28
S5	62.9%	95.9%	21	58.7%	78.0%	19	60.8%	87.0%	40
S6	26.4%	83.1%	5	-	-	-	26.4%	83.1%	5
TOTAL	46.7%	90.1%	43	60.0%	80.5%	50	53.4%	85.3%	93

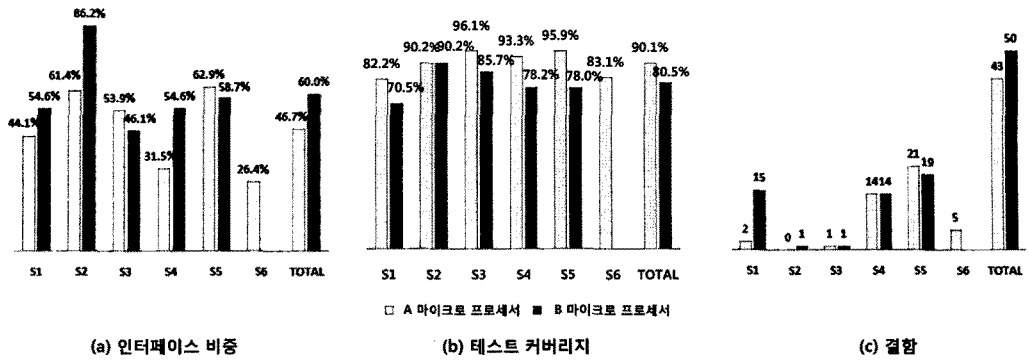

그림 6 Justitia 테스트 결과

페이스 결함을 소프트웨어 개발 단계의 조기에 발견 할 수 있을 것이라고 추정된다.

본 사례 연구는 제안한 임베디드 소프트웨어 테스트 방안이 현실적으로 구현가능하고 실용화될 수 있음을 보이고 있다.

5. 임베디드 소프트웨어 테스트 도구 비교 분석

본 절에서는 제안한 Justitia와 대표적인 임베디드 소프트웨어 테스트 도구들인 IBM Rational의 Test Real Time[8], Mertrowerks의 CodeTEST[9], TestQuest의 TestQuest Pro[10]와 PolySpace의 PolySpace[11]를 테스트 레벨, 테스트 활동, 테스트 환경, 테스트 기법 측면에서 비교 분석한다.

(1) 테스트 레벨

테스트는 단위, 통합, 시스템, 인수 테스트의 테스트 레벨이 존재한다. 테스트 레벨이 다양한 이유는 테스트 레벨 별로 테스트하는 목적과 발견할 수 있는 결함의 종류가 다르기 때문이다. 앞서 언급했듯이 임베디드 소프트웨어는 특정 소프트웨어만 독립적으로 테스트 하기 힘들기 때문에 대부분 시스템 레벨에서 소프트웨어 와 하드웨어의 경계 구분 없이 테스트가 이루어진다.

이러한 임베디드 소프트웨어 특성은 소프트웨어의 잠재적 결함을 발견하기 어렵고, 결함을 발견하더라도 발생 위치와 결함 원인에 대해 파악하기 매우 힘들다. 그렇기 때문에 이들 계층들 사이의 상호작용인 인터페이스를 테스트하는 통합 테스트는 매우 중요하다. 즉, 인터페이스는 통합된 계층내의 소프트웨어나 하드웨어 중 어느 부분에 결함이 존재하는 지 식별하는 매우 중요한 기준이다. 표 4와 같이 현재 대다수의 임베디드 소프트웨어 테스트 도구들은 임베디드 시스템의 메모리 사용 현황, CPU 처리 속도의 성능 테스트와 제품 기능 테스트에 집중되어 있다. 소프트웨어 - 소프트웨어, 소프트웨어 - 하드웨어 사이의 인터페이스를 테스트하는 통합 테스트 레벨의 지원은 미흡하며, Justitia는 이들 이질적 계층 사이의 통합 테스트를 위한 임베디드 소프트웨어 인터페이스 테스트를 지원한다.

(2) 테스트 활동

테스트는 테스트 계획, 테스트 케이스 설계, 테스트 수행, 테스트 결과분석의 일련의 활동들로 이루어진다. 특히, 테스트 케이스를 설계하고, 결함이 있는지 테스트가 얼마나 이루어졌는지를 판단하는 결과 분석은 기술집약적인 활동으로 테스트에 대한 많은 경험과 테스트

표 4 임베디드 소프트웨어 테스트 도구[8-11]

제 품	자동화 범위									핵심 특징
	테스트 레벨*				테스트 활동**			테스트 환경	테스트 기법	
	CT	SST	IT	ST	TD	TE	TR			
Test RealTime	O	O	X	X	X	O	X	시뮬레이터 타겟	상태 기반 테스트	메모리 프로파일링, 성능 프로파일링, 프로세스 프로파일링, 트레이스, 코드 커버리지
CodeTEST	O	O	X	O	X	O	X	시뮬레이터 이뮬레이터 타겟	소스 기반 테스트	메모리 프로파일링, 성능 프로파일링, 트레이스, 코드 커버리지
TestQuest Pro	O	X	X	O	X	O	X	타겟 기반 테스트	시나리오 기반 테스트	UI (User Interface) 테스트, 성능 테스트, 부하 테스트, 운영체제 테스트, 상호운용성 테스트
PolySpace	O	X	X	X	O	X	X	비실행 환경	소스 기반 테스트	메모리 프로파일링, 코드 커버리지
Jutitia	X	O	O	X	O	O	O	이뮬레이터	인터페이스 테스트	인터페이스 커버리지, 실행 경로 정보, 메모리 프로파일링, 결함 정보

(* 테스트 레벨 - CT: Component Test, SST: Software System Test, IT: Integration Test, ST: System Test, ** 테스트 활동 - TD: Test Design, TE: Test Execution, TR: Test Result)

대상에 대한 상당한 도메인 지식을 필요로 한다. 이에 반해 테스트 수행 활동은 테스트 케이스를 반복적으로 실행하는 노동집약적 활동인데, 표 4의 비 실행 기반의 PolySpace를 제외하면 모두 반복적이고 기계적인 테스트 수행을 자동화하고 있다. 그러나, 테스트 대상에 대한 지식과 테스트 경험이 요구되는 테스트 설계 및 결과 분석 자동화는 미흡하다. Justitia는 특히 테스트에 대한 이해와 경험이 미흡한 개발자를 위해 인터페이스를 중심으로 한 테스트 케이스 선정, 테스트 드라이버 생성, 테스트 수행과 결과 분석의 모든 테스트 활동을 자동화함으로 개발자의 테스트 활동을 강화하고 있다.

(3) 테스트 환경

임베디드 소프트웨어 테스트에서 무엇보다 해결해야 할 문제는 변화하는 테스트 환경을 지원하는 것이다. 표 4의 테스트 도구들은 하드웨어에 의존성을 해결하기 위해 시뮬레이터, 이뮬레이터, 타겟 하드웨어에서 단계적으로 테스트될 수 있도록 테스트 수행 환경을 지원하고 있다. Justitia는 시스템 레벨의 테스트를 수행하기 이전의 소프트웨어 검증을 위한 테스트를 지원하며 이를 위해 이뮬레이터의 디버깅 기능을 테스트 활동에 통합한 자동화 도구이다. 현재 스크립트 개발하여 이뮬레이터를 자동 실행되도록 하는 방법은 흔히 사용되고 있지만, Justitia와 같이 이뮬레이터 사용에 특정 테스트 기법을 적용한 자동화는 없다. Justitia는 임베디드 소프트웨어 개발자에게 실질적인 도움을 주기 위하여 그들이 디버깅 목적으로 사용하는 이뮬레이터의 중단점, 모니터링과 같은 디버깅 기능을 임베디드 소프트웨어의 핵심 테스트 위치인 인터페이스를 자동 테스트할 수 있도록 승화시켰다.

(4) 테스트 기법

많은 테스트 도구들이 CodeTEST나 PolySpace와 같이 소스 코드 기반의 화이트 박스 테스트 기법이나 Test RealTime과 같이 실시간 소프트웨어 테스트를 위한 상태 기반의 테스트 기법을 지원한다. 그러나, 이러한 기법은 임베디드 소프트웨어 개발자가 겪는 테스트의 어려움을 해결하기엔 부족함이 있다. Justitia는 개발자 고충을 덜기 위해 임베디드 소프트웨어의 핵심 테스트 위치인 이질적 계층의 인터페이스를 테스트하고 결함 여부를 판정하는 인터페이스 테스트 기법을 자동화하고 있다.

6. 결론 및 향후 연구

본 논문은 임베디드 소프트웨어가 실제 하드웨어에 탑재되어 제품이 되기 이전에 이뮬레이터를 이용하여 소프트웨어를 테스트하는 방안과 이를 구현한 자동화 도구 Justitia v2.0을 제안하였다. 본 논문을 통해 해결한 임베디드 소프트웨어 테스트 이슈는 다음과 같다.

첫째, 본 논문은 임베디드 소프트웨어 인터페이스를 새로운 테스트 기준으로 하는 테스트 방안을 제안하였다. 임베디드 소프트웨어 인터페이스는 다양한 이질적 계층의 소프트웨어와 하드웨어가 밀접히 결합되어 상호작용하는 임베디드 소프트웨어 특징을 반영한 테스트 기준이다.

둘째, 본 논문은 임베디드 소프트웨어 인터페이스 테스트를 위한 자동화 방안을 정의하고, 이에 따라 테스트 케이스 선정, 테스트 드라이버 생성, 테스트 수행과 테스트 커버리지 분석의 핵심 테스트 활동을 자동화한 테스트 도구 Justitia를 개발하였다.

셋째, 본 논문은 개발자가 까다롭게 여기는 테스트 활동을 이뮬레이터의 단순한 디버깅 기능과 통합하는 형태로 자동화하고, 이를 통해 개발자의 테스트 활동을 강화하였다.

현재 Justitia는 모바일 용 ARM 프로세서 기반 BSP의 소프트웨어 테스트에 적용되고 있다. 또한, 임베디드 소프트웨어 인터페이스 테스트 방안이 효과적임을 분석하기 위해 대표적인 화이트박스 테스트 기준과 테스트 커버리지, 결함 발견율, 결함 위치 적합율을 비교하는 실험을 진행 중이다.

참 고 문 헌

[1] Rumeson, P., Andersson, C., and Host, M., "Test processes in software product evolution - a qualitative survey on the state of practice," Software Maintenance and Evolution: Research and Practice, vol.15, pp. 41-59, 2003.

[2] *A* Co., Ltd.,Technical Report of SQA Evaluation Model and Guide for Embedded Software Test Process Improvement 2004.

[3] Gal-Oz, S., Isaacs, M. V., "Automate the big bottleneck in embedded system design," IEEE Spectrum, pp. 62-66, 1998.

[4] Yoo, S., Jerraya, A. A., "Introduction to hardware abstraction layers for SoC," Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE) '03, pp. 10,336-10,337, 2003.

[5] Sung, A., Choi, B., and Sin, S., "An interface test model for hardware-dependent software and embedded OS API of embedded system," Computer Standard & Interface, vol.29, pp. 430-443, 2007.

[6] *A* Co., Ltd.,, Technical Report of Emulation Test Tool for Linux and ARM based Embedded Software Interface Coverage, 2007.

[7] Earnshaw, R., ARM Procedure Call Standard for the ARM Architecture, ARM, 2005.

[8] IBM Rational's Test RealTime, www.ibm.com

[9] MetroWerks's CodeTEST, www.metrowerks.com

[10] TestQuest's TestQuest Pro, www.testquest.com
[11] PolySpace's PolySpace, www.polyspace.com

서 주 영

1993년 이화여자대학교 컴퓨터학 학사 2001년 이화여자대학교 컴퓨터학 석사 2004년~현재 이화여자대학교 컴퓨터학 박사과정. 관심분야는 임베디드 소프트웨어 테스트, 테스트 자동화, 소프트웨어 프로세스 개선

최 병 주

1983년 이화여자대학교 수학과 학사. 1988년 Purdue Univ. 전산학 석사. 1990년 Purdue Univ. 전산학 박사. 1995년~현재 이화여자대학교 컴퓨터학과 교수. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 소프트웨어 및 데이타 품질 측정, 소프트웨어 프로세스, 임베디드 시스템 테스팅, 서비스 기반 아키텍처