임베디드 소프트웨어를 위한 테스트와 디버깅 연계 자동화 방안
(Automated Coordinator between Testing and Debugging of Embedded Software)

최 유 나 † (YooNa Choi) 서 주 영 †† (JooYoung Seo)

최 병 주 ††† (ByoungJu Choi)

요 약 임베디드 소프트웨어는 하드웨어 소프트웨어의 결합력이 매우 높기 때문에 전체 시스템에 대한 사용 시나리오 기반의 블랙박스 테스트가 주로 수행된다. 본 논문은 블랙박스 테스트로 발견된 결함에 대한 디버깅이 쉽지 않음에 착안하여 테스트와 디버깅 활동 연계를 지원하는 자동화 방안을 제안한다. 제안하는 방안은 테스트 결과로부터 결함 원인과 위치 추적이 가능한 디버깅 전략을 수립하는 방안과 이를 기반으로 이뮬레이터 환경에서 자동 수행되는 테스트 스크립트 자동 생성하는 방안으로 구현된다.

키워드 : 임베디드 소프트웨어 테스트, 디버깅, 동적 메모리 결함

Abstract *Generally, due to the strong coherence between* embedded software and hardware or peripheral software, embedded software is tested by using black-box test based on user scenario for the whole system. This paper suggests the method to coordinate between testing and debugging under consideration for difficulties on solving out the defects detected from black-box test. First of all, from test result analysis, it builds up the debugging strategies enable to trace the locations of the defect's causes. And along with the strategies, it implements the generator of test scripts to be performed on the emulator environment. Through these steps, it can coordinate embedded software testing and debugging activities.

Key words : Embedded Software Test, Debugging, Dynamic Memory Defects

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(NIPA-2010-(C1090-0903-0004)

· 이 논문은 제36회 추계학술발표회에서 '임베디드 소프트웨어 시스템 테스트에서의 동적 메모리 결함 해결 연계 자동화 연구'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 이화여자대학교 컴퓨터공학과
yoona@ewhain.net

†† 정 회 원 : 이화여자대학교 컴퓨터공학과
jyseo@ewhain.net

††† 종신회원 : 이화여자대학교 컴퓨터공학과 교수
bjchoi@ewha.ac.kr

논문접수 : 2009년 12월 21일
심사완료 : 2010년 2월 18일

Copyright©2010 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 제16권 제5호(2010.5)

1. 서 론

최근 사회 전반의 중요한 기반을 이루고 있는 임베디드 시스템의 소프트웨어 요구사항 다양화와 기능 복잡화에 따라 임베디드 시스템 내의 소프트웨어 품질에 대한 관심과 요구도 증대되고 있다. 임베디드 산업 현장에선 시스템 결합의 80% 이상이 하드웨어보다는 소프트웨어에 의한 것이라는 보고[1]도 있으며, 소프트웨어 품질이 전체 시스템의 품질로 직결되기 때문에 이에 대한 테스트 활동이 강화되고 있는 추세이다.

일반적으로 하드웨어와 소프트웨어들의 결합력이 매우 높다는 임베디드 소프트웨어의 특성 때문에 소프트웨어만 독립적으로 테스트하기 쉽지 않고, 전체 시스템에 대한 사용 시나리오 기반의 블랙박스 테스트가 주로 수행된다. 하지만, 블랙박스 테스트는 결함 발견 후 그 원인과 위치를 파악하는 데에 어려움이 많기 때문에, 이러한 테스트 환경에서 발견된 결함은 디버깅하는 데에 많은 시간과 비용이 소모된다.

특히 시스템 테스트의 경우, 일반적으로 소프트웨어 내부 구조를 이해하지 못하는 제3자 테스터에 의해 테스트 활동이 수행되기 때문에 테스트 시 발견된 결함의 디버깅을 도와 줄 정보를 관련 개발자에게 추가로 제공하는 것은 매우 어려운 일이다. 시스템 테스트 레벨에선 해당 시스템의 기능 및 비기능적인 행위를 제품레벨에서 검증하는 것을 목적으로 하기 때문에 테스트 활동들이 결함 발견 후 복구를 위한 디버깅 활동을 연계할 수 있도록 고려되고 있지 않다. 테스터는 소스코드가 아닌 실행코드를 전달받아 테스팅을 하기 때문에 소프트웨어의 내부 구조를 파악하기 어렵고, 개발자는 자신이 구현한 코드 이외의 영역에 대한 정보가 부족하기 때문에 디버깅 활동은 개발자의 개인적인 능력과 경험적 판단에 매우 의존적일 수 밖에 없다.

본 논문은 임베디드 소프트웨어의 시스템 테스트 단

계에서의 테스트와 디버깅 두 공정의 괴리(gap)를 축소 시키기 위해 테스트를 통해 발견된 결함에 대한 위치를 추적하고 원인 식별을 돕는 테스트 자동 생성 방안을 제공함으로써 테스트와 디버깅 공정을 연계하여 전체 개발 공정을 단축할 수 있는 자동화 방안을 제안한다.

이 논문의 구성은 다음과 같다. 2장에서 테스팅과 디버깅 활동에 대해 살펴본 후, 3장에서 본 논문이 제안하는 테스팅과 디버깅 공정 연계 자동화 방안에 대해 기술한다. 4장과 5장에선 제안하는 방안의 적용효과와 결론을 기술한다.

2. 테스팅과 디버깅 활동

임베디드 소프트웨어의 높아진 관심만큼 테스팅에 대한 많은 연구들과 적용 사례들이 늘어나고 있는[1-3] 반면, 디버깅에 대한 연구는 소스코드레벨의 분석 기법이나[2,3] 디버거 기술에 대한 연구[3,4]가 대부분이다. 테스팅을 통해 발견한 결함을 해결하기 위한 디버깅으로의 연계 방안에 대한 연구는 더욱이 미비한 실정이다. 특히 단위 테스트 레벨에서 소스 코드 변형을 통해 테스트를 수행하고 디버깅을 유도하는 연구들이 많이 진행되어 왔지만[3,4], 실행 코드를 입력으로 하는 시스템 테스트 레벨에서의 디버깅 연계에 관한 연구는 미흡하다.

테스팅은 테스터가 결함에 의한 실패를 확인하고, 결함의 존재를 인지하는 활동인 반면, 디버깅은 개발자가 결함의 원인을 확인하고, 코드를 수정하고, 결함이 올바르게 고쳐졌는가를 확인하는 개발 활동이다. 또한, 시스템 테스트의 경우 실제 타겟 시스템을 실시간 환경에서 시나리오 기반의 블랙박스 테스트가 수행되는 반면, 디버깅의 경우 이뮬레이터 환경에서 소스 코드 기반의 화이트박스 방식으로 수행된다. 이렇게 명확히 구분되는 활동임[4]에도 테스팅과 디버깅은 매우 강하게 결합된 일련의 개발 활동이다. 다시 말해, 디버깅은 테스팅 없이는 불가능하고, 테스팅 활동은 발견된 결함을 해결하는 디버깅 작업이 없이는 그 활동이 무의미하게 된다.

테스터는 테스트 수행을 통해 결함을 발견하고 나면 테스트 산출물(Test Deliverable)로써 결함 보고서(Bug/Defect Report)를 개발자에게 전달하게 된다. 결함 보고서란 하나의 결함에 관련한 여러 증상들 또는 오류들을 설명하는 기술적 문서이다. 개발자는 결함보고서를 바탕으로 결함을 해결하기 때문에, 테스터가 얼마나 디버깅에 도움을 줄 수 있는 정보를 제공하느냐가 성공적인 디버깅에 매우 중요하다. 다음의 IEEE Std.829에서 정의한 결함 보고서를 받은 개발자는 자신의 개발 코드에서 보고된 오류를 일으킨 결함을 찾기 위한 디버깅 활동에 들어간다. 해당 오류를 발견한 테스트 케이스에 따라 결함 재현을 위한 새로운 테스팅이

결함 보고서 식별자(Test Incident report identifier) 문제 요약(Summary) 결함 설명(Incident descriptions) 결함이 끼치는 영향력(Impact)

일어나지만, 임베디드 시스템의 경우 그 결함이 발생된 시점의 시스템 상황과 상태 값들이 일치하지 않으면 항상 재현되지도 않을 수 있고, 재현했다고 하더라도 결함을 수정하기 위하여 모니터링해야 하는 위치 설정과 테스트 결과를 성공/실패(Pass/Fail)로 판단하기 위한 기준 및 판단 근거들이 개발자의 개인적 능력에 의존하고 있다. 이 때문에 실제 결함을 해결하기 위한 코드 수정보다도 어디에 어떤 원인들이 있을 수 있고, 실제 존재하는 지에 대한 결함 원인 추적에 개발자가 투자하는 시간적, 물리적 비용이 너무 많이 들기 때문에 개발자에게는 결함 디버깅은 부담이 큰 활동이 된다.

3. 테스팅과 디버깅 연계 자동화 방안

본 논문은 테스트와 디버깅 공정의 자동 연계를 위해 시스템 테스트의 결과 중 오류 재현 정보를 TRACE32 이뮬레이터를 자동 실행하는 스크립트로 제공함으로써 결함 판별과 동시에 즉시 디버깅할 수 있도록 테스팅과 디버깅 공정 연계를 자동화 방안을 제안한다.

제안하는 테스팅과 디버깅의 연계 자동화 방안은 다음과 같은 3단계가 요구된다. 1단계에는 결함 유형에 따른 디버깅 전략을 수립한다. 디버깅 전략은 결함 별로 결함 원인으로 의심되는 위치인 디버깅 영역, 그 위치에서 살펴봐야 할 모니터링 변수 및 값, 그리고 어떠한 기준을 가지고 해당 결함 테스트의 결과를 판정할 것인지에 대한 성공/실패 판단 기준이다. 2단계에는 앞서 세운 디버깅 전략에 맞게 테스트 스크립트 템플릿을 구축한다. 3단계에서는 테스트 스크립트 자동 생성 알고리즘을 개발하는 것이다.

3.1 연구 적용의 범위

본 논문은 동적 메모리 결함 11유형을 JTAG 디버그 로직의 디버깅 기법을 이용하여 디버깅에 연계하는 방안을 제시한다.

3.1.1 동적 메모리 결함

본 논문에서는 동적 메모리 결함 11가지 유형을 해결하는 디버깅을 타겟으로 연계 방안을 소개하고 있다. 그러나 제안하는 방안은 동적 메모리 결함에 국한된 내용이 아니라 다른 종류의 결함에도 동일한 방법으로 테스팅과 디버깅을 연계 시킬 수 있다.

동적 메모리 결함들은 정적 테스팅을 통해서는 발견되기 힘든 결함 유형으로서 실행 기반의 동적 테스팅을 통해서만 발견할 수 있는 결함들이며, 소스 코드 분석만

을 통해선 문제의 원인을 찾기 힘든 결함들이 대부분이다. 특히 동적 메모리 결함 중에는 전체 시스템을 다운시킬 수 있는 위험수준이 높은 결함들이 존재한다.

3.1.2 JTAG 디버그 로직

본 논문에서 제안하는 방안은 호스트 PC환경에서 JTAG기반의 이뮬레이터인 TRACE32를 활용하여 디버깅 할 수 있도록 테스트 스크립트를 자동 생성하는 방안이다. 그러나 이는 특정 이뮬레이터에 국한하지 않는다.

JTAG(Joint Test Action Group)은 디버깅 및 테스트를 목적으로 칩(Chip) 내에 구현된 국제 규격(IEEE 1149.1)이며, 타겟 시스템과의 인터페이스가 간단하여 모든 칩에 공통으로 적용 가능하다. TRACE32 이뮬레이터는 타겟 시스템 CPU의 JTAG 포트를 이용하여 JTAG이 제공하는 디버깅 신호를 받아들여 디버깅을 가능하게 하는 디버거이다.

3.2 결함 유형에 따른 디버깅 전략 수립

시스템에 탑재되는 임베디드 소프트웨어가 실제 런타임 환경에서 가질 수 있는 메모리 결함들의 유형을 파악하고, 각 유형마다의 결함 원인을 파악하여 디버깅 영역과 결함 판별 기준 등의 디버깅 전략을 구축한다.

3.2.1 디버깅 영역(BreakPoint Candidates)

디버깅 영역은 결함 원인 분석을 위해 어떤 위치들을 모니터링 할 것인가에 해당한다. 시스템을 시나리오 기반으로 테스트한 후 결함이 발생하면, 결함의 발생 위치는 파악할 수 있지만, 결함을 해결하는 디버깅으로 연계하기 위해서는 결함의 위치뿐만 아니라 해당 결함이 발생하기 전의 상황을 파악하여 결함발생 원인이 될만한 위치들을 찾아내야 한다. 결함의 발생 원인으로 선정된 위치들이 곧 디버깅 수행 시 중단점(Breakpoint)이 된다.

Duplicate Free결함의 경우, 동일한 메모리 영역을 재해제시 발생하는 결함이기 때문에, 당연히 동일한 메모리 영역을 해제(FREE)했던 두 위치를 보아야 할 것이며, 그 메모리 영역이 해제되기 전 가장 최근의 할당(ALLOC)은 어디서 일어났는지도 볼 수 있다면 결함 해결하는데 이해가 수월할 것이다. 그래서 실행 순서에 따라, 할당-해제-해제(ALLOC-FREE-FREE)가 일어난 3개의 영역이 테스트해야 할 위치인 디버깅 영역이 된다. 11개의 동적 메모리 결함 유형 마다의 결함 원인 위치 즉, 디버깅 시 중단점들을 정리한 것은 다음 표 1과 같다.

3.2.2 모니터링 변수/값(Monitoring Variables)

테스트 수행 중에 어디에 멈추어서 디버깅할지 결정이 되었다면 그 위치에서 어떤 값 또는 변수들을 모니터링하여 최종적으로 결함이 있음의 성공/실패를 판정할지를 파악해야 한다. 이러한 값 또는 변수들이 곧 모니터링 변수(Monitoring Variables)가 된다. 실제로 타겟에서 소프트웨어 프로그램이 실행할 때, 인스트럭션들이

표 1 동적 메모리 결함 유형 별 중단점

메모리 결함 유형	중단점(Breakpoint)
Leakage	ALLOC
Zero alloc	ALLOC
Fail alloc	ALLOC
Illegal free	ALLOC-FREE
Null pointer free	ALLOC-FREE
Duplicate free	ALLOC-FREE-FREE
Null pointer access	ACCESS
Free pointer access	ALLOC-FREE-ACCESS
Invalid pointer access	ALLOC-ACCESS
Out of bound access	ALLOC-ACCESS
Collision	ALLOC-ALLOC-ACCESS

수행되면서 주고받는 값들은 대부분 레지스터 셋(Register Set)을 통해서 수행된다.

예를 들어, MIPS 하드웨어 플랫폼의 타겟 시스템을 테스트 하는 경우 동일한 메모리 영역을 할당-해제-해제(ALLOC-FREE-FREE)한 경우 발생한 Duplicate Free 결함을 해결하기 위한 모니터링 변수들이 실제 어떤 레지스터 셋에 매핑되는지를 살펴보면 다음과 같다. Rn은 n번 레지스터 값을 의미한다.

- 할당(ALLOC) 위치 - 할당 크기(R4), 할당한 메모리 주소, 할당 후 리턴 주소(R31)
- 해제(FREE) 위치 - 해제된 메모리 주소, 해제 횟수를 세기 위한 FreeCount변수

위와 같이 결함 별로 디버깅 영역이 되는 위치에서 모니터링 해야 하는 값이나 변수들이 분산되지 않고 몇 개의 미리 정의된 레지스터들로 제한되기 때문에 디버깅 시 결함 발생에 대한 자동 판정이 가능해진다.

3.2.3 성공/실패 판단 기준(PASS/FAIL Criteria)

디버깅 전략으로 정해진 중단점 위치에서 특정 값이나 변수들을 모니터링하고, 테스트의 성공/실패 판단 기준을 세워 놓으면 자동적으로 결함 발생에 대한 성공/실패를 판정할 수 있게 된다.

예를 들어, Duplicate Free 결함의 경우, 해제 횟수를 가리키는 FreeCount변수가 첫 메모리 해제 위치에서 0값을 갖고 있다가 1값으로 증가하게 되고 두 번째 해제시, 1값 이상이면 두 번째 해제임을 인식하고 실패로 판정하게 된다.

3.3 테스트 스크립트 템플릿 구축

테스트 스크립트 템플릿의 전체 구조는 기능적으로 크게 '설정 관련 템플릿, GUI(Graphic User Interface) 템플릿, 결함 유형별 템플릿'의 3가지 유형으로 분류하여 개발하여 제공한다. 설정 관련 템플릿은 실행 파일(.exe)과 심볼 파일(.pdb)을 다운로드하고 초기화 시킨다. GUI 템플릿은 사용자가 TRACE32를 통해 테스트

시, 보다 편하고 쉬운 인터페이스를 제공하는 템플릿이 다. 그리고 실제 결함 유형 별로 그림 1과 같은 기본 수행 구조를 기반으로 3.2절에서 구축된 디버깅 전략 별로 이뮬레이터에서 자동 수행 가능한 테스트 스크립트 템플릿을 개발한다.

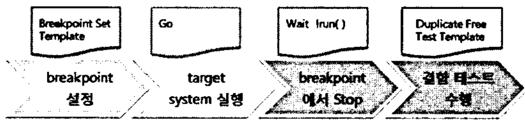

그림 1 결함 테스트 스크립트 내부의 수행 구조

테스트 스크립트를 실행시키면 디버깅 전략에 따라 선정된 중단점이 자동 설정되고, 타겟 시스템이 실행되다가, 중단점 위치를 만나면 실행을 멈춘 후, 해당 결함에 대한 테스트를 수행하는 순서로 테스트가 자동적으로 수행된다.

3.4 테스트 스크립트 자동 생성 알고리즘

결함의 원인이 되는 위치를 추적하기 위하여 TRACE32 이뮬레이터를 구동시켜 발견된 메모리 결함을 해결하기 위하여 테스트 스크립트를 생성하는 테스트 스크립트 자동 생성 모듈(Test Script Generator)을 개발하였다. 테스팅과 디버깅 연계 방안으로서 테스트 스크립트가 자동 생성되는 과정을 그림 2에서 보여주고 있다.

동적 메모리 결함 디버깅에 연계하기 위하여 테스트 수행을 통해 얻은 메모리 로그를 입력으로 한다. 메모리 로그는 테스트 대상이 되는 어플리케이션이 실행 하는 순서대로 메모리 액션이 일어날 때 남겨진 로그로서, 실행시간, 메모리 액션(할당, 해제, 또는 접근), 메모리 주소, 크기 등의 정보를 담고 있다. 생성된 메모리 로그로부터 디버깅 전략에 따라 결함 발생원인 위치가 될만한 디버깅 영역에 해당하는 위치를 추출하여 해당 결함 별로 중단점 후보들을 구성한다. 결함이 발생된 메모리 주소와 같은 위치에서 일어난 모든 메모리 액션들이 일어난 위치가 중단점으로 모두 선정되었다면, 미리 세워두었던 디버깅 전략에 의해 생성된 스크립트 템플릿과 함께 디버깅으로 연계된 테스트를 수행하게 하는 테스트 스크립트가 자동적으로 생성된다.

4. 적용 및 분석

임베디드 소프트웨어 시스템 테스팅을 통해 발견된 결함을 해결하는 디버깅으로의 과정으로의 연계 자동화 방안으로 제안한 테스트 스크립트 자동 생성 모듈을 적용해 본 사례를 기술하고, 결과를 분석한다.

4.1 적용 대상

테스트 스크립트 자동 생성 모듈을 통한 디버깅에 연계 방안을 A, B 두 프로젝트로 나누어 적용하였다. 프로젝트 A의 경우, 실제 before-market 차량용 AVN (Audio Video Navigation) 시스템에 메모리 결함 유형 11가지를 고의적으로 삽입한 테스트 모듈을 구현하여 스크립트 자동 생성 모듈을 통해 결함의 원인 위치에 잘 도달하여 성공/실패를 판정하는지 확인하였다. 프로젝트 B의 경우, 총 14대의 AVN 시스템들(S1-S14)을 테스트 대상으로 하였으며, 시스템들의 세부 명세는 다음 표 2와 같다.

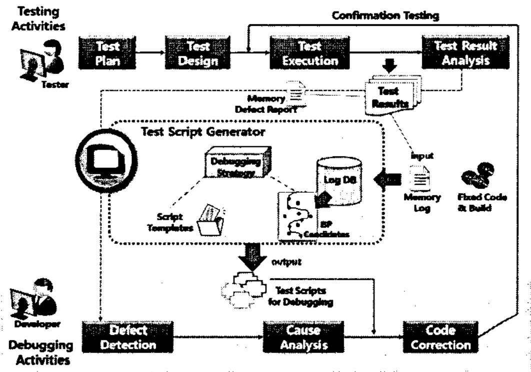

그림 2 테스팅과 디버깅 연계 과정

표 2 테스트 적용대상 시스템 명세

OS	WINCE5.0						WINCE6.0				QNX			
HW	ARM				MIPS		ARM			MIPS	SH4			
System	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}

4.2 적용 결과

프로젝트 A에서 삽입한 11가지 유형의 결함들을 모두 발견하였고, 각 결함들에 대하여 테스트 스크립트 자동 생성 모듈로 자동 생성한 테스트 스크립트를 TRACE32로 실행시킨 결과, 테스트 스크립트 수행을 통해 모든 결함의 원인이 되는 위치에 찾아 가서 미리 선정해 놓은 모니터링 값들을 토대로 성공/실패 판단 기준에 따라 자동으로 실패를 성공적으로 판정하였다.

그림 3은 프로젝트 A를 통해 발견한 결함들 중에 Free Pointer Access 결함에 대해 자동 생성된 테스트 스크립트를 TRACE32를 통해 수행한 결과이다.

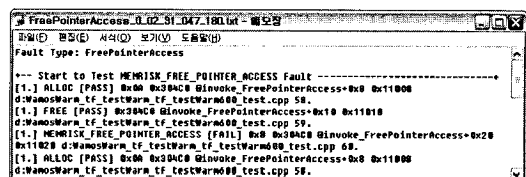

그림 3 테스트 스크립트 수행 후 판정 결과

메모리 영역 0×304C0을 0×0A크기만큼 할당하고 동일한 0×304C0을 해제한 뒤 0×8크기만큼 동일한 메모리 영역을 접근하여 결국, Free Pointer Access 결함으로 판정한 것을 볼 수 있다. 더불어 d:\amos\arm_tf_test\arm_tf_test\arm600_test.cpp소스코드의 경로와 할당은 58번째 라인, 해제는 59번째 라인에서 접근은 60번째 라인에서 일어난 것임을 정확히 보여주고 있다.

프로젝트 B의 14대의 AVN 시스템을 테스트한 결과 발견한 동적 메모리 결함들은 다음 표 3과 같으며, 결함 별 원인 위치 추적 기능을 포함하는 테스트 스크립트 자동 생성 모듈을 통해 적용하였다.

표 3 before-market 시스템 테스트 적용 결과

OS		WINCE5.0							WINCE6.0				QNX			
HW		ARM					MIPS		ARM			MIPS	SH4			
Memory Faults		S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	TOTAL
1	LEAKAGE	3	0	0	0	0	0	0	1	3	0	1	26	2	2	38
2	ZERO ALLOC	0	5	2	6	0	0	0	0	0	5	1	0	0	0	19
3	FAIL ALLOC	0	0	0	0	0	0	3	0	0	0	0	0	0	0	3
4	ILLEGAL FREE	0	1	1	1	0	0	0	0	0	1	2	0	0	0	6
5	NULL POINTER FREE	15	14	9	11	15	1	1	0	0	5	7	0	0	0	73
6	DUPLICATE FREE	6	19	25	29	5	4	5	6	0	5	4	3	1	0	112
7	NULL POINTER FREE	0	0	0	0	0	10	0	0	0	1	1	0	0	0	12
8	FREE POINTER ACCESS	30	52	13	17	2	12	0	0	0	3	2	0	0	0	131
9	INVALID POINTER ACCESS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	OUT OF BOUND ACCESS	2	0	0	2	2	0	0	0	0	0	0	0	0	1	7
11	COLLISION	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	TOTAL	56	91	50	66	24	26	9	7	3	22	12	29	3	3	401

4.3 기대 효과 분석

테스팅과 디버깅 공정을 긴밀하게 연계시키는 과정을 자동화함으로써 전체 개발 공정을 단축시키는 효과를 보일 수 있다. 결함 발생 후 결함을 테스터가 개발자에게 제기 시, 기존의 결함 보고서와 자동 생성된 테스트 스크립트는 수행 동시에 결함원인이 될만한 위치들을 분석하여 소스코드 기반으로 알려주기 때문에 보고받은 결함을 탐지하고, 결함원인을 파악하기 위해서 분석하는 프로세스를 줄일 수 있는 것이다.

디버깅 활동의 책임자인 개발자 입장에서도 테스트 스크립트를 통해서 자동적으로 결함 유형 별 원인 위치들이 중단점으로 설정되어 있어 보다 결함 원인에 보다 빠르고 정확히 접근한다는 측면에서 유리하다. 테스트 스크립트를 함께 전달 받은 개발자 입장에서는 결함 보고를 받았을 때, 어느 곳을 수정해서 결함을 해결해야 하는지가 이미 테스트 스크립트를 통해 정해져 있기 때문에, 또한 그 위치에서 성공/실패 판단 기준에 의해 자동 판정하는 기능까지 있어 결함 디버깅 테스트가 간편해지고 근본적인 결함 원인에 도달하는 정확성 또한 높아진다. 테스터 입장에서도 테스트 수행 후 테스트 결과 분석 시에 자동으로 테스트 스크립트를 생성하여 전달할 수 있으므로 보다 객관적인 결함 정보를 개발자에게 전달할 수 있다.

5. 결 론

임베디드 소프트웨어는 전체 시스템에 대한 블랙박스 테스트로 발견된 결함에 대한 디버깅이 쉽지 않음을 보았다. 또한 테스터가 결함 원인과 원인 위치에 대한 디버깅 정보를 제공해 주지 못하고, 개발자 역시 결함해결의 디버깅을 위해 개인적인 능력과 판단에 의존하기 때문에 결함 원인 추적에 시간적, 물리적 비용이 너무 많이 들어왔다.

본 논문은 테스트와 디버깅 활동 연계를 지원하는 자동화 방안을 제안하였다. 먼저 테스트 결과 분석과 디버깅 전략 수립을 통해 결함 원인 위치 추적을 자동화하였다. 그리고 JTAG 기반의 TRACE32 이뮬레이터 환경에서 추적된 결함 원인 위치들에 대하여 자동적으로 테스트를 수행할 수 있는 테스트 스크립트를 제공하는 모듈을 구현하였다. 이렇게 임베디드 소프트웨어 테스팅과 디버깅 공정을 연계시킴으로써 전체 개발공정을 단축시키는 효과를 갖는다. 본 논문에서는 동적 메모리 결함에 촛점을 두어 기술하였으나 제안하는 방안은 동적 메모리 결함에 국한된 내용이 아니라 다른 종류의 결함에도 동일한 방법으로 테스팅과 디버깅을 연계 시킬 수 있다. 현재 시스템 단계의 주요 테스트인 동적 성능 테스트와 같은 비기능 테스트에도 확장 적용하고 있으며 성능 병목의 위치와 원인을 찾는데 적용 효과를 보고 있다.

참 고 문 헌

[1] J. Y. Seo, A. Y. Sung, B. J. Choi, S. B. Kang, "Automating Embedded software Testing on an Emulated Target Board," Proc. of the Second International Workshop on Automation of Software Test, 2007.

[2] Andrea Arcuri, "On the Automation of Fixing Software Bugs," International Conference on Software Engineering(ICSE), pp.1003-1006, 2008.

[3] A. Zeller, "Automated debugging: Are we close?" IEEE Computer, vol.34, no.11, pp.26-31, 2001.

[4] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," IEEE International Conference on Automated Software Engineering (ASE), pp.30-39, 2003.