

사례연구: 테스트 자동화 도구를 이용한 임베디드 소프트웨어의 블랙박스 테스트

A Case Study of Black-Box Testing for Embedded Software using Test Automation Tool

국 문 요약

본 논문은 전형적인 임베디드 소프트웨어 중 하나인 Temperature Controller(TC)의 블랙박스 테스트에 대한 사례연구를 제시한다. 앞선 연구를 통해 개발한 테스트 자동화 도구인 TEST를 이용하여 여러개의 상용 TC에 대한 테스트를 수행하였다. 이를 통해 임베디드 시스템의 테스트 결과에 대한 통계적 분석결과를 제시하고, 임베디드 소프트웨어가 갖는 소프트웨어 오류의 속성을 정의하였다. 또한 아래와 같은 내용을 제시하고 있다. (a) 테스트 프로세스 안에서 상대적으로 오류의 리뷰를 위해 많은 시간이 요구되기 때문에 테스트 케이스 우선순위 기법이 요구된다. (b) 임베디드 소프트웨어의 오류는 크게 세가지 타입으로 정의될 수 있다. (c) 임베디드 시스템의 하드웨어 구성 복잡도는 소프트웨어 오류에 영향을 미친다. (d) 소프트웨어 오류는 해당 소프트웨어의 취약지점에 분포되는 특성이 있다. (e) 테스트 활동을 통해 소프트웨어 오류의 개수를 줄일 수 있다. 본 논문에서 제시되는 결과를 통해 임베디드 소프트웨어의 블랙박스 테스트에 대한 여러 문제를 고찰하는데 도움을 줄 수 있을 것이다.

키워드: 테스트 자동화 도구, 임베디드 소프트웨어, 임베디드 시스템, 블랙박스 테스트, 시스템 테스트

영 문 요약

This paper presents a case study of the Black-Box testing for Temperature Controller (TC) which is one of the emblematic embedded systems. A test automation tool, TEST, was developed and some kinds of TCs were tested using the tool. We present statistical analysis for the test results obtained from the automated testing and define the properties of the some typical software bugs in embedded system. The main contributions of this research are following: (a) A test case prioritization technique is needed because the review phase in testing takes a long time (b) We classify the embedded software bugs into the three categories (c) the complexity of the system configuration has a vivid effect on the software defects (d) the software defects are distributed in vulnerable points of the software and (e) testing activities reduce the number of the software defects. The results can be useful in the asymptotic study of test case prioritization.

keywords: test automation tool, embedded SW, embedded system, black-box test, system test

1. 서 론

임베디드 소프트웨어 시장 규모가 확대됨에 따라, 제품의 품질과 신뢰성 확보는 소프트웨어 개발 프로세스 안에서 매우 중요한 문제로 대두되고 있다. 2004년 세계적으로 유명한 모바일 회사의 경우 몇가지 소프트웨어 결함으로 인해 추가적인 애프터서비스를 진행하여 막대한 금전적 손실을 지불했다. 또한 제품 수명 주기가 더욱 짧아짐으로 인해 더욱 많은 소프트웨어 결함이 내재될 가능성이 높아지고 있다. 이를 위해 임베디드 소프트웨어의 품질 보증에 대한 노력이 요구되고 있다[1].

많은 대기업과 중소기업에서는 각각의 개발 단계에서 소프트웨어 결함을 줄이기 위한 노력을 하고 있다. 이 경우 가장 일반적으로 사용되는 접근 중 하나는 테스트 자동화 도구를 이용하여 개발된 제품이 요구사항에 만족하는지의 여부를 테스트 하는 것이다. 테스트 자동화 도구는 매뉴얼 테스트와 비교하여 보다 정확하며, 테스트로 하여금 보다 편리한 테스트 환경을 제공한다. 그러나 기존에 존재하는 많은 테스트 자동화 도구의 경우 대부분이 유닛 테스트에 적합한 도구이며, 블랙박스 테스트와 연관된 도구의 경우 해당 제품에 적용하기 위해 많은 추가적인 노력이 요구되기도 한다. 이와 같은 문제는 자동화 테스트 자체가 불가능하다는 결론으로 귀결되어 해당 테스트팀 내에서 기존의 매뉴얼 테스트를 고수하게 하였다. 또한 임베디드 시스템의 하드웨어 구성은 매우 다양하여, 시스템 테스트 관점에서 개발되는 시스템에 탑재된 하드웨어와 완전하게 호환하는 테스트 자동화 도구를 찾는 것은 거의 불가능한 것이 현실이다. 반대로 하나의 테스트 자동화 도구를 여러 종류의 시스템에 수정없이 적용하는 것 또한 불가능하다[2,3,4].

위에서 언급한 상황이나 제약으로 인하여, 임베디드 시스템 개발사의 개발 프로세스 내에서는 기존 테스트 자동화도구를 제약적으로 사용할 수 밖에 없었으며, 심지어는 사용하지 못하는 상황이 일어나기도 하였다. 이를 위해 개발 중인 시스템에 적합한 테스트 자동화 환경을 구축하기 위해 기존 시스템의 대대적인 변화를 가하거나, 혹은 새로운 시스템을 개발할 때마다 새로운 테스트 자동화 도구를 개발하였다[5,6,7].

본 논문에서는 임베디드 시스템에 적합한 테스트 자동화 도구를 개발한 내용을 제시하고 있다. 또한 개발된 테스트 자동화 도구를 이용하여 여러개의 상용 제품에 대해 실시한 테스트 결과를 소개하고, 결과 자료를 바탕으로 분석적으로 접근 할 수 있는 테스트 관련 이슈에 대한 접근법을 제시하고자 한다. 본 논문의 구성은 아래와 같다. 먼저 2절에서는 테스트 타겟이 되는 Temperature Controller(TC)와 테스트를 위해 개발된 테스트 자동화 도구의 설계와 구현에 대해 소개한다. 3절에서는 다양한 테스트 결과를 통해 소프트웨어 테스트에서 자동화 도구를 이용하는 것에 대한 장점과 최근 임베디드 소프트웨어 테스트 분야에서 제시되는 몇가지 이슈에 대한 접근법을 제시한다. 마지막으로 4절에서는 결론과 향후 과제를 기술하였다.

2. 관련 연구

2.1 Temperature Controller

본 논문에서 System Under Test(SUT)로 이용한 Temperature Controller(TC)는 여러개의 센서와 스위치 입력을 기반으로 여러개의 액추에이터를 제어하는 전형적인 임베디드 시스템이다. 실제로 TC는 외기온 센서, 내기온 센서, 습도 센서, 포토센서 등으로부터 외부 환경 인자를 받아들여, 풍량과 풍향 등을 결정하게 된다. 또한 OFF 스위치에 대한 사용자의 입력을 바탕으로 여러개의 액추에이터의 동작 유무를 판단하게 된다. 아래 그림1은 TC의 하드웨어 구성을 보여준다. 또한 TC는 내부 소프트웨어의 동작 방식에 따라 FATC(Full Automatic Temperature Controller)와 MTC(Manual Temperature Controller)로 구분된다. MTC는 외부 액추에이터의 동작 방식이 사용자 선택에 의해서만 작동하며, 이에 따라 내부 소프트웨어 또한 상대적으로 간단하다. FATC는 사용자가 선택한 타겟 온도를 실현하기 위해 여러개의 액추에이터 동작이 소프트웨어적으로 구현되어 있는 제품으로, 내부 소프트웨어의 복잡도가 상대적으로 MTC 보다 크다고 할 수 있다.

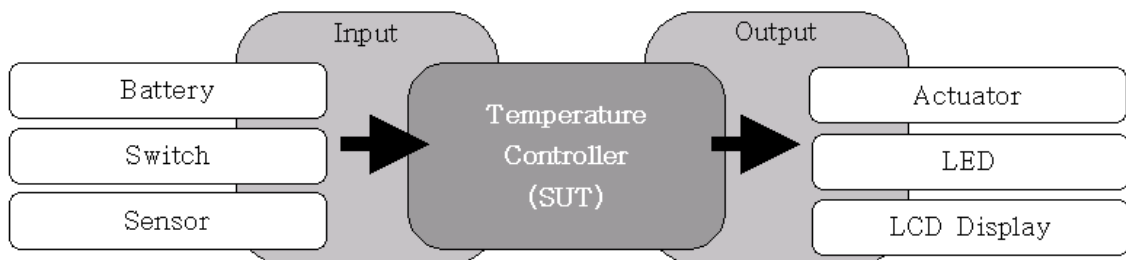


그림 1 Temperature Controller

따라서 본 논문에서는 다수의 FATC 제품에 대한 시스템 테스트를 수행한 결과를 제시하고 있다. 시스템 테스트를 위해 아래 그림에서 Battery, Switch, Sensor 등으로 구성된 테스트 케이스를 생성하고, SUT로부터 생성되는 Actuator, LED, LCD Display 출력값을 수집하여 소프트웨어 오류 존재 여부를 판단하였다.

2.2 테스트 자동화 도구

소프트웨어 테스트를 위해 테스트 자동화 도구를 사용하는 주된 이유는 매뉴얼 테스트가 가진 여러 가지 단점을 극복하기 위해서이다. 테스트 케이스를 작성하고, 이를 많은 수의 테스터가 실행한 후 결과를 리뷰하는데 요구되는 시간 비용이 매우 크다. 앞선 연구에서 자동화 테스트와 비교하여 매뉴얼 테스트가 두배 이상의 시간과 비용을 요구한다는 사실

을 제시하고 있다[8]. 또한 테스트의 정확성 측면에서도 주어진 테스트 케이스의 실행에 있어서 정해진 타이밍을 매뉴얼 테스트에서는 보장할 수가 없으며, 이로 인해 소프트웨어 오류를 찾았다 하더라도 재현하지 못하는 경우가 많이 발생한다.

테스트 자동화 도구의 성능은 주어진 테스트 프로세스의 각 단계를 얼마만큼 정확하게 자동화하느냐에 달려 있다. 즉, 테스트 케이스의 생성과 테스트 케이스의 실행, 그리고 Actual Result의 생성과 Expected Result와의 자동 비교등의 기능에 테스트 자동화 도구가 이와 같은 항목을 지원할수록 매뉴얼 테스트와 비교해 많은 장점을 제공할 수 있다. 또한 자동화되어 생성되는 보고서는 효과적인 오류 추적을 위해 효율적인 구조를 지녀야 하며, 이를 위해 많은 연구자들이 효과적인 보고서 템플릿 형태를 제시하고 있다 [9, 10, 11].

본 논문에서는 임베디드 소프트웨어의 블랙박스 테스트를 수행하기 위해 TEST (Testing-stand for Embedded Software Testing)을 개발하였다[12]. TEST는 유한머신 기계(Finite State Machine) 모델링 방법을 기반으로 스펙을 기술하고, 이에 대한 모델 체크를 지원하며, 테스트 케이스 자동 생성과 실행, 그리고 Expected Result와 Actual Result 간의 자동 비교[13,14] 및 보고서 생성[15,16]에 이르기까지 테스트 프로세스 전체에 걸친 자동화를 지원하고 있다. 또한 SUT에서 발생하는 출력값에 대해 SUT와 상호 작용하는 아이템의 피드백 전송을 위해서 가상환경관리자를 구현하여 해당 액추에이터에서 SUT로 보내는 피드백 값을 자동으로 전송하는 환경을 구축하였다[17].

아래 그림2는 TEST를 이용한 테스트 수행 절차를 나타내고 있다. TEST를 이용하기 위해서는 가장 먼저 SUT에 대한 모델링 과정을 수행해야 한다. TEST는 모델링된 자료와 테스터의 지식(Knowledge)을 바탕으로 반자동 테스트 케이스를 생성한다. 생성된 테스트 케이스는 자동으로 실행되며, 이때 모델링 자료와 테스트 케이스 입력값으로 Expected Result를 실시간으로 생성한다. 생성된 Expected Result는 SUT의 실제 출력값인 Actual Result와 자동으로 비교된 후 오류 여부를 테스터에게 보고하게 된다.

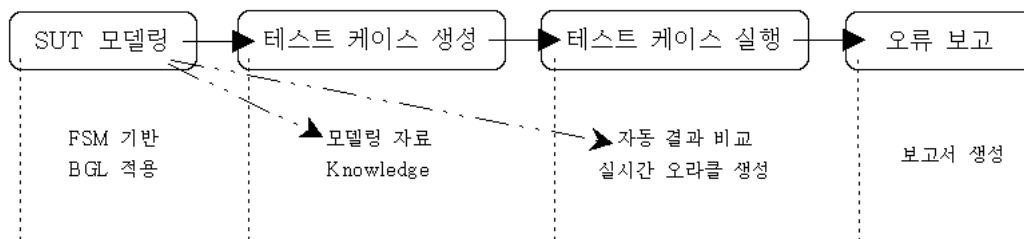


그림 2 TEST를 이용한 테스트 절차

3. 테스트 수행 결과

본 절에서는 개발된 테스트 자동화 도구인 TEST를 이용하여 상용 TC에 대한 테스트 결과를 소개하고, 이 결과를 바탕으로 도출할 수 있는 몇가지 사항을 소개한다. 2절에서 언급한 것과 같이 SUT 대상인 TC는 두가지 종류도 구분되나, MTC 타입의 경우 소프트웨어 기능이 단순하여 소프트웨어 오류가 보고되지 않기 때문에, 본 절에서는 7개의 DATC

타입의 TC에 대한 테스트 결과를 중심으로 소프트웨어 오류의 특징과 테스트 결과의 활용 방안 등을 기술하고자 한다.

그림 3 테스트 결과 분석을 위한 원시 데이터

CODE for MN	MN (Module Name)	CODE for OVN	OVN (Output Variable Name)
M1	기동풍량제어	O1	ACT_Intake_AIM
M2	난방기동제어Blower	O2	ACT_Intake_FED
M3	난방기동제어Mode	O3	ACT_Intake_OUT
M4	냉방기동제어	O4	ACT_Mode_AIM
M5	변수 SET	O5	ACT_Mode_FED
M6	변수 Set2	O6	ACT_Mode_OUT
M7	A/C Auto제어	O7	ACT_Temp_AIM
M8	Ambient Table	O8	ACT_Temp_FED
M9	Ambientcoefficient설정	O9	ACT_Temp_OUT
M10	AmbientSensor보정	O10	CTL_AC_OUT

표 1 Module Name과 Output Variable Name 참조표

그림 3은 테스트 수행 결과에서 여러 특징을 찾기 위한 원시 데이터를 보여주고 있다. 즉 하나의 소프트웨어 오류에 대해서 오류의 원인 및 실제 임베디드 시스템의 출력 (Code for OVN¹) 항목과 내부 로직(Code for MN²) 항목) 등을 연관시켜 정리하였다. 아래 그림에서 하나의 오류에 대해 최대 3개 까지의 외부 아이템의 동작에 이상작동을 유발 시킨다는 가정을 바탕으로 테스트 결과를 정리하였다. 예를 들어, 에어컨 출력과 관련된 로직에 오류가 존재할 경우 에어컨 출력을 담당하는 액추에이터의 동작은 물론 해당 출력 상태를 사용자에게 통보하는 LED 부분에도 문제가 발생하기 때문이다.

본 절에서 위의 그림3과 같이 정리한 데이터와 테스트 시간 비용 및 시스템의 하드웨어 구성 복잡도 등을 바탕으로 분석 가능한 여러개의 항목에 대해 아래 세부절에서 기술한다. 먼저 3.1절에서는 TEST를 이용한 테스트에 요구되는 시간 비용을 각 테스트 단계에 따라 분석하였다. 3.2절에서는 발견된 소프트웨어 오류의 원인을 몇가지 종류로 정의하였고, 3.3절에서는 하드웨어 구성 복잡도와 소프트웨어 오류 발생 빈도의 관계에 대해 언급하였다. 3.4절에서는 임베디드 시스템 관점에서 시스템 출력과 소프트웨어 오류의 연관성에

1) Code for OVN :Output Variable Name에 대한 코드로 임베디드 시스템에서 외부로 내보내는 출력 변수에 대한 참조를 위해 사용된 O1~O53까지의 참조 번호를 의미함. 표1 참조.

2) Code for MN :Module Name에 대한 코드로 스펙을 기준으로 임베디드 시스템 내부에 존재하는 로직을 정의하고 이에 대한 참조를 위해 사용된 M1~M58까지의 참조 번호를 의미함. 표1 참조.

대해 기술하였으며, 3.5절에서는 TEST 이용한 지속적인 임베디드 소프트웨어 테스트를 통해 얻어진 효과에 대해 소개한다.

3.1 테스트 수행 시간

본 절에서는 1년간 진행된 총 7개의 DATC 타입의 TC에 대한 테스트에 요구되는 시간 비용에 대해 기술한다. 각각의 시스템 복잡도에 따라 단일 시스템의 소프트웨어 테스트에 요구되는 시간은 2주에서 1달 정도의 시간이 요구된다. 이때 TEST에 의해 실행된 테스트 케이스의 개수는 정도의 차이는 있지만 대략 5 만여개 정도이다. 아래 그림은 각 테스트 단계에 따른 수행시간을 보여주고 있다.

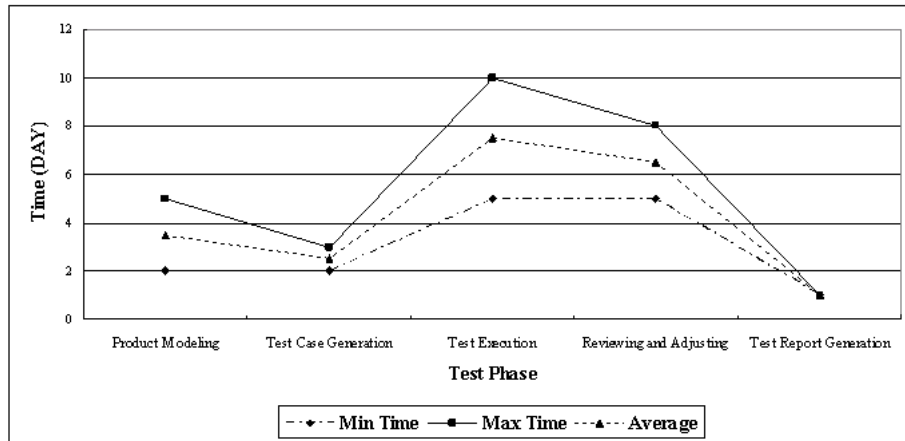


그림 4 테스트 단계에 따른 수행 시간 비교

위 그림 4에서 테스트 케이스의 실행과 실행 이후 발견된 오류를 리뷰하고 오류로 판단하는데 걸리는 시간이 상대적으로 오래 걸림을 알 수 있다. 즉 테스트 프로세스의 시간 비용 절감을 위해서는 오류의 발견 시점이 테스트 케이스 실행 단계의 초반이나 또는 후반이나에 따라서 오류 발견 시점부터 시작되는 리뷰 단계를 케이스 실행 초기에 시작되고, 케이스 실행 후반에 가서는 오류가 발생되지 않기를 기대할 수 밖에 없다.

3.2 소프트웨어 오류 분류

본 절에서는 테스트 결과를 바탕으로 임베디드 소프트웨어의 오류를 분류한 방법을 제시한다. 우리는 본 논문에서 테스트 결과 발견된 소프트웨어 오류에 대해 프로그래밍 오류, 스펙 불이해 오류, 스펙 오류 등으로 세가지로 나누어 구분하였다.

프로그래밍 오류(Programming Error)란 전통적인 소프트웨어 오류이며, 프로그래머의 실수에 의해 야기된 오류를 의미한다. 프로그래밍 오류는 발견된 오류 중 74%에 해당하는 비중을 차지하였다. 예를 들어 아래 표2과 같이에서 “>=”이 들어갈 자리에 “>”로 입력

할 경우 해당 경계값의 처리와 관련된 소프트웨어 오류를 나타낸다.

```

if ( iVar >= 255) ----- ① correct code
if ( iVar > 255) ----- ② error code
    
```

표 2 프로그래밍 오류 예제

스펙 불이해 오류(Specification Misunderstanding Error)란 시스템 스펙을 프로그래머가 잘못 이해하거나 잘못 해석하여 발생한 오류이다. 스펙에서 기술한 로직이 복잡하고 어려운 경우 개발자 스스로 이를 해석하는 과정에서 발생된다. 자연어로 기술된 스펙을 개인적 관점에서 해석할 때 발생되며, 이러한 오류의 경우 스펙 기술자와 개발자 간에 테스트 보고서를 바탕으로 여러 차례의 논의가 진행된 후에야 오류로 분류가 되어 그 만큼 많은 리뷰 시간을 요구하게 된다. 예를 들어 여러개의 입력이 하나의 출력에 영향을 미치는 경우 입력들 간의 우선순위에 대한 프로그래밍에 오류가 존재하는 경우가 이에 포함된다. 스펙 불이해 오류는 전체 오류 중 14%에 해당하였다.

스펙 오류(Specification Error)란 스펙 자체에 오류가 포함된 경우이다. 개발자에게 제공된 스펙의 다의적 해석 가능성이 존재하거나, 로직간의 충돌 시에 우선 순위 등을 명확히 하지 않은 경우에 발생한 오류를 스펙 오류로 구분한다. 이와 같은 스펙 오류는 테스트 결과 중 전체의 12%에 해당하는 발생빈도를 보였다.

이와 같이 3가지 형태로 임베디드 소프트웨어 오류를 구분하는 이유는 해당 SUT의 오류 발생 빈도를 낮추기 위한 방법을 결정하기 위해서이다. 즉 예를 들어 스펙 오류의 발생 빈도가 현저하게 높을 경우에는 개발 프로세스 상에서 스펙을 기술하는 과정 안에서 좀더 명확하고 정확하게 기술하는 방법을 적용해야 할 것이며, 스펙 불이해 오류가 많이 존재하는 경우 스펙 기술자와 개발자 간의 협의 과정에 대해 개발 프로세스 내에서 좀더 많은 시간 비용을 투자할 필요가 있기 때문이다.

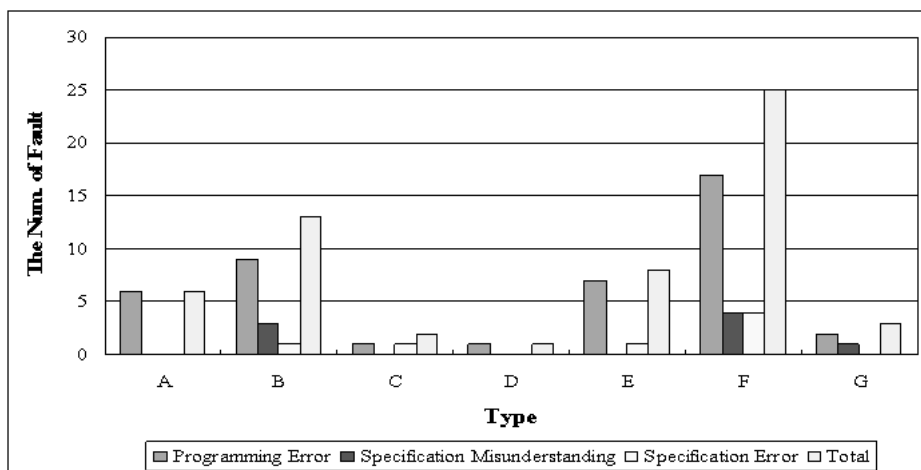


그림 5 각 시스템과 오류 구분에 따른 소프트웨어 오류 분포

추가로 위의 그림5는 발견된 오류를 각 제품과 소프트웨어 오류 분류에 따라 나누어

보여주고 있다. 예를 들어 F 제품의 경우 TEST를 이용한 시스템 테스트를 통해 총 25개의 소프트웨어 오류를 발견한 것을 의미한다. 세부적으로 F 제품의 프로그래밍 오류는 17개이고, 스펙 불이해 오류는 4개, 스펙 오류는 4개가 발견되었음을 보여준다.

3.3 하드웨어 구성 복잡도와 소프트웨어 오류

본 절에서는 임베디드 시스템의 입출력과 관련된 하드웨어 구성 복잡도와 소프트웨어 오류 간의 관계를 분석한 것을 제시한다. 아래 그림은 SUT인 해당 시스템에 연결된 액추에이터나 센서의 개수와 발견된 소프트웨어 오류의 개수 사이의 관계를 보여주고 있다. 아래 그림에서 보여지는 것과 같이 센서나 액추에이터와 같은 시스템에 연결된 엔드 아이템(End Items)의 개수가 증가함에 따라서 발견되는 소프트웨어 오류의 개수도 증가함을 알 수 있다. 즉, 이러한 결과를 바탕으로 전통적인 소프트웨어 테스팅에서 이용하는 소프트웨어 복잡도 평가에서 코드 라인수 (LOC, Line of Code)를 이용하는 것과 같이 엔드 아이템의 개수를 복잡도의 척도로 이용할 수 있다.

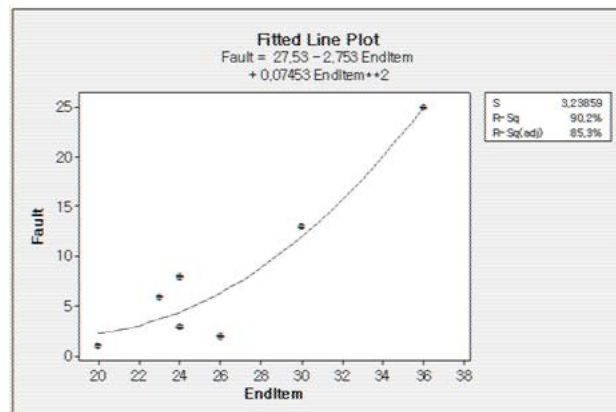


그림 6 엔드 아이템과 소프트웨어 오류 간의 관계

우리는 잠재적인 소프트웨어 오류 개수를 엔드 아이템의 개수로 예측할 수 있다. 이러한 사실은 테스트 수행의 종료 시점을 예측하는 기준으로 사용될 수 있다. 그림 6과 그림 7은 실제 테스트 결과를 기반으로 MINITAB을 이용하여 생성된 결과이다. 우리는 엔드 아이템 개수와 소프트웨어 오류 사이를 다항 회귀 분석(Polynomial Regression Analysis)을 진행하였다. 이를 통해 수식1에 기술된 것과 같은 회귀 방정식(Regression Equation)을 찾을 수 있었다(단, $S=3.23859$, $R-Sq=90.2\%$, $R-Sq(adj) = 85.3\%$, $F_{Regression} = 18.38$, $P_{Regression} = 0.010$). 이와 같은 회귀 방정식을 이용하여, 현재 진행하는 시스템의 엔드 아이템 개수를 입력하여, 잠재 소프트웨어 오류 개수를 예측한다. 예를 들어, 해당 시스템의 양산 일정에 밀려 테스트 수행 시간이 정해진 시간보다 적을 경우에 이미 회귀방정식을 통해 예측된 잠재 소프트웨어 오류를 찾았다면, 현재 테스트 수행을 중지하더라도 시스템의 복잡도 관점에서 과거 기준과 비교하여 비슷한 수준의 품질을 보장받는다고 할 수 있다.

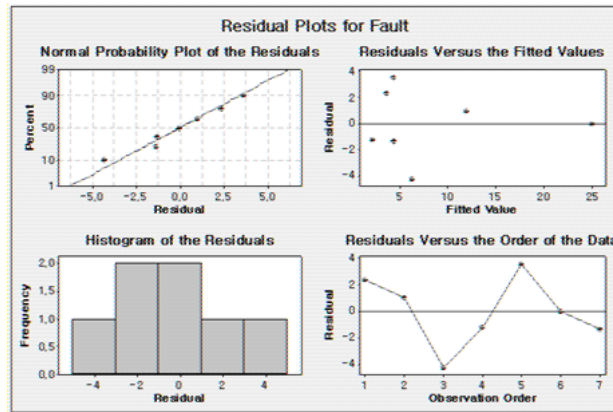


그림 7 소프트웨어 오류에 대한 Residual Plots

$$Y = 0.0745X^2 - 2.7535X + 27.531$$

수식 1 엔드 아이템과 소프트웨어 오류 사이의 회귀 방정식

3.4 시스템 출력과 소프트웨어 오류

본 절에서는 임베디드 시스템에서 외부 입력의 결과로 만들어진 출력값이 해당하는 엔드 아이템과 발견된 소프트웨어 오류 사이의 관계를 분석한다. 테스트에 SUT로 활용된 7개의 TC는 각각 유사한 기능을 포함하고 있으며, 그에 따라 연결된 하드웨어 엔드 아이템 또한 거의 일치한다. 아래 그림 8은 발견된 소프트웨어 오류가 영향을 미치는 시스템 출력 간의 관계를 매핑한 결과이다. 또한 그림 9는 발견된 오류와 관련하여 프로그램 내에 어느 위치에서 오류가 발생한 것인지를 내부 로직과의 매핑을 통해 분석한 결과이다.

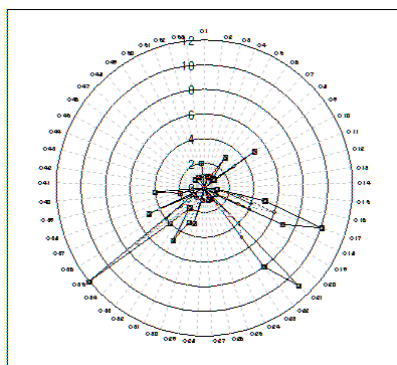


그림 8 시스템 출력에 따른 SW 오류 분포

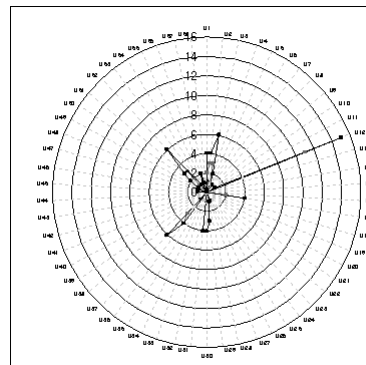


그림 9 내부 로직에 따른 SW 오류 분포

그림 8에서 보여진 바와 같이, 특정 출력에 소프트웨어 오류가 집중되어 있는 사실을 알 수 있다. 그림에서 O17, O21, O35는 다른 출력과 비교하여 현저하게 많은 오류 발생

빈도를 보여준다. 또한 그림 9에서 M12의 경우 다른 내부 로직과 비교하여 보다 많은 오류 발생 빈도를 나타내고 있다. 여기서 O17과 O21, O35는 실제로 Defog 관련 액추에이터와 LED 등을 의미하며, M12는 Defog 제어 로직이다.

결과적으로 SUT인 TC 내부에서 Defog 제어³⁾와 관련된 기능에서 빈번하게 소프트웨어 오류가 발생하는 경향이 있다는 사실을 찾을 수 있다. Defog 제어는 시스템 내에서 상대적으로 취약한 기능임을 알 수 있으며, 이는 이와 관련된 테스트 케이스를 조기에 실행할 경우 그만큼 소프트웨어 오류를 발견할 확률이 높음을 의미한다.

소프트웨어 오류 발생 확률에 대한 테스트 케이스 순위화 기법에 대해 많은 연구가들이 활발한 활동이 진행 중이다. Hema Srikanth 등은 고객-할당 우선순위와 요구사항 복잡도 등을 기반으로 가중치를 계산하는 방식을 제시하였다[18]. 그러나 이와 같은 방식은 고객과 개발자에 의해 제시되는 가중치를 테스트에 이용하므로 테스트에 이용되는 데이터가 데이터 생성자 중심이라는 한계를 가지기 때문에, 순서화 과정이 특정 전문가 의존적이 되는 단점을 가진다. 따라서 이 방식은 비전문가 집단에 의해 우선 순위가 부여될 경우 오히려 순서화를 적용하지 않은 경우보다 비효율적인 결과를 초래할 수 있다. 또한 Sebastian Elbaum 등은 함수 호출 개수에 따라 테스트 케이스를 순서화하고, 또한 코드의 변화를 이용하여 회귀 테스트(Regression Testing)에서 변화된 부분에 가중치를 부여하는 방식을 제안하였다[19]. 그러나 이 방식은 화이트 박스 테스트에만 국한된 방식으로 블랙박스 기반의 시스템 테스트에는 적합하지 않다.

본 절을 통해 임베디드 시스템의 특성 상 특정 하드웨어에 대한 제어가 어려우며, 이는 곧 해당 소프트웨어의 취약성으로 간주될 수 있음을 알 수 있었다. 동일한 도메인 내에서는 하드웨어 구성이 유사하기 때문에 이를 이용하여 앞서 인용한 두가지 방식과 다른 과거 이력 기반 테스트 케이스 순위화 기법에 대한 연구에 이와 같은 과거 이력 기반 접근법이 활용될 수 있다[20]. 또한 이를 통해 4.1절에서 언급한 테스트 케이스 실행에 따른 리뷰 시간의 조기 종료를 기대할 수 있다.

3.5 테스트 활동과 소프트웨어 오류 감소

본 절에서는 시스템의 품질 향상을 위해 수행하는 테스트 활동 결과 해당 SUT의 오류 발생 빈도의 변화를 살펴본다. 그림 10은 단일 기업(실제 TC 생산을 담당하는 개발업체 기준)에서 생산하는 TC 4개에 대한 TEST를 이용한 테스트 결과 발견된 오류의 개수를 보여주고 있다. 그림에서 기준이 된 SUT의 순서는 시간을 기준으로 정렬한 결과이다. 즉, 테스트 자동화 도구인 TEST 도입 후 첫 번째 적용하는 TC에서는 13에 달하는 소프트웨어 오류가 발견되었고, 이후 점차 감소하여 네 번째 TC에 대한 테스트에서는 총 3개의 소프트웨어 오류만이 발견되었음을 보여준다. 이는 그동안 매뉴얼 테스트를 통해 발견되지 못한 소프트웨어 오류가 테스트 자동화 도구의 도입으로 인하여, 보다 정확하고 보다 많은 테스트 케이스를 실행 결과 새로운 오류로 발견되어 개발업체는 이 부분에 대한 개선 활동을 하는 것으로 해석될 수 있다.

3) Defog 제어 : 창문 서리/습기 제거를 위해 스위치 ON, 또는 System ON 시에 Aircon 제어.

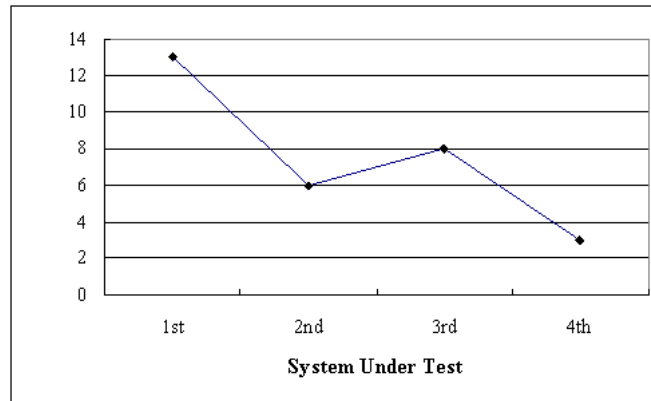


그림 10 단일 기업 내 소프트웨어 오류의 감소

4. 결론 및 향후 과제

본 논문에서는 전형적인 임베디드 시스템 중 하나인 TC에 대한 블랙박스 기반 시스템 테스트를 위해 구현된 테스트 자동화 도구인 TEST를 이용한 테스트 결과를 소개하였다. TEST는 1년간 매뉴얼 테스트를 수행한 7개의 양산 제품에 대한 추가적인 자동화 테스트를 수행하였으며, 단일 TC를 기준으로 최고 25개의 소프트웨어 오류를 발견하였다.

발견된 소프트웨어 오류 결과를 바탕으로 테스트 자동화 도구인 TEST를 이용한 테스트 수행 시간을 각각의 테스트 단계에 따라 분석하였고, 발견된 오류를 세가지 기준으로 구분하여 프로그램 내부의 오류 뿐만이 아니라 개발 프로세스의 개선을 통한 제품 품질 향상의 근거를 제시하였다. 또한 단위 시스템에 연결된 하드웨어 구성 복잡도와 오류 사이의 관계를 회귀방정식으로 정의하여 잠재적인 소프트웨어 오류 평가에 활용될 수 있도록 하였으며, 시스템 내의 소프트웨어 구성 부분 중 특정 영역에 취약성이 존재하는 사실을 근거로 하는 테스트 케이스 순위화 기법의 적용 필요성을 제안하였다. 마지막으로 발주사에서 진행한 적극적인 테스트 노력이 개발업체에도 영향을 미쳐 테스트 수행 전 단계에서의 오류 비율이 감소한다는 사실을 파악할 수 있었다.

그러나 본 논문에서 테스트 결과를 활용한 여러 가설은 특정 도메인에 국한되어 적용 가능하다는 한계를 가지며, 이를 극복하는 것은 향후 과제이다. 하지만 도메인이 변경되었을 경우라 하더라도 해당 테스트 결과를 본 논문에서 접근한 방식을 통해 잠재 소프트웨어 오류를 예측하는 등의 방법론은 충분히 활용될 수 있을 것으로 기대하고 있다.

5. 참고 문헌

1. W. Grant Ireson, Clyde F. Coombs, and Richard Y. Moss, Handbook of Reliability Engineering and Management, McGraw-Hill Professional, 1995.

2. Boris Beizer, Black-Box Testing, John Willey & Sons, 1995.
3. B. Beizer, Software Testing Techniques, 2nd Ed., International Thomson Computer Press, 1990.
4. 장영숙, 여기대, 이현동, “Manual과 Automated 테스트에 대한 사례 연구”, 정보과학회 추계학술발표 논문집(2), 2003.
5. Bart Broekman and Edwin Notenboom, Testing Embedded Software, Addison Wesley, 2003.
6. Kelvin Rodd, Practical Guide to Software System Testing, K.J. Ross & Associates Pty. Ltd, 1998.
7. OVUM, Software Testing Tools, OVUM, 1999.
8. M. Sowers, Software Testing Tools Summary, Software Development Technologies Inc. White Paper, 2002.
9. L. Hayes, The Automated Testing Handbook, Software Testing Institute, 1995.
10. M. Fewster, D. Graham, Software Testing Automation: Effective use of test execution tools, ACM Press, Addison Wesley, 1999.
11. E. Dustin, J.Rashka, and J. Paul, Automated Software Testing, Addison Wesley, 1999.
12. *Changhyun Baek*, Seungkyu Park, and Kyunghee Choi, TEST: An Effective Automation Tool for Testing Embedded Software, WSEAS Trans. On Information Science & Applications, Issue8, Volume2, August 2005.
13. 김범모, *백창현*, 장중순, 정기현, 최경희, 박승규, "임베디드 SW의 블랙박스 테스트를 위한 검증 모듈의 디자인 및 구현", VOL.31 NO.02, 한국정보과학회 추계학술대회, 2004.
14. Douglas Hoffman, Using Test Oracles in Automation, Pacific Northwest Software Quality Conference, 2001.
15. 전형인, *백창현*, 박승규, “임베디드 소프트웨어의 효율적인 오류추적을 위한 테스트 수행 보고서”, Vol32, KICS Fall Conference, 2005.
16. IEEE-SA, IEEE829: IEEE Standard for Software Test Documentation, IEEE-SA Standards Board, 1998.
17. 김범모, *백창현*, 장중순, 정기현, 최경희, 박승규, “임베디드 소프트웨어 테스트를 위한 가상 환경 관리자의 디자인 및 구현”, 2005 한국컴퓨터종합학술대회 논문집, KCC 2005, 2005.
18. Hema Srikanth and Laurie Williams, Requirements-Based Test Case Prioritization, GHC2004, 2004.
19. Sebastian Elbaum, Gregg Rothreml, and Satya Kanduri, Selecting a Cost-Effective Test Case Prioritization Technique, 185~210, Vol12(3), Software Quality Journal, 2004.
20. *백창현*, 태상원, 신승훈, 박승규, “임베디드 소프트웨어를 위한 과거 이력 기반 테스트 케이스 순위화 기법, 2006 한국컴퓨터종합학술대회 논문집(C), KCC2006, 2006.

