실시간 이동형 내장 소프트웨어 시험 도구의 구조 설계

(Architecture Design for Real-time Mobile Embedded Software Test Tools)

김 상 일 † 이 남 용 †† 류 성 열 †††
(Sang Il Kim) (Nam Yong Lee) (Sung Yul Rhew)

요 약 소프트웨어의 생산성을 높이고 신뢰성 있는 실시간 이동형 내장 소프트웨어를 개발하기 위해서는, 소프트웨어를 실시간으로 정확하게 분석하고 시험 검증할 수 있는도구가 필요하다. 이러한 도구는 기본적으로 소스코드 기반의 화이트박스 시험 기능, 실시간 시스템 모니터링과 실행 제어 기능을 필요로 하며, 향후 다양한 시스템 환경과의 연동을 고려하여 재사용성과 이식성을 높일 수 있도록 설계되어야 한다. 본 논문에서는 실시간 이동형 내장 소프트웨어를 시험하기 위한 시험검증 도구의 기능적 요구사항을 식별하고, 식별된 요구사항을 바탕으로 시험검증 도구에 적합한 구조를 설계하였다. 또한 시험검증 도구의 확장성과 이식성 제고를 위해 각 기능에 대한 구체적인 구현 기술과 기법을 제시하였으며, 이들 기능의 설계에 디자인 패턴을 적용하였다.

키워드 : 소프트웨어 테스팅, 임베디드 소프트웨어 테스팅, 소프트웨어 아키텍처, 임베디드 소프트웨어 아키텍처, 모바일 소프트웨어 테스팅, 실시간 임베디드 소프트웨어

Abstract A tool for analyzing and testing software in real-time is required for the efficient development of highly reliable real-time mobile embedded software. This tool requires various technologies, such as source code based white-box test and real-time system monitoring and control. The tool also should be designed to improve reusability and portability by considering the interaction with other kinds of real-time system. This paper identifies and analyzes the functional requirements for the test tool on real-time mobile embedded software and suggests an adequate tool architecture based on the collected requirements. It also suggests the specific implementation technology and architecture design pattern to support the tool's expandability and portability.

Key words : Software Testing, Embedded Software Testing, Software Architecture, Embedded Software Architecture, Mobile Software Testing, Real-time embedded software

1. 서 론

임베디드 소프트웨어 개발의 비중이 점차 높아져 감에 따라 산업체 전반에 걸쳐 임베디드 소프트웨어의 중요성이 매우 중요한 위치를 차지하고 있다. 임베디드 시스템 개발의 핵심은 임베디드 소프트웨어 개발 기술이며, 실시간 이동형 내장 소프트웨어는 일반 소프트웨어와는 달리 실시간성, 고신뢰성, 소용량 메모리, 저전력을 요구하는 품질 특성을 갖는다. 따라서 실시간 이동형 내장 소프트웨어의 생산성을 높이고 신뢰성 있는 소프트웨어를 개발하기 위해서는 소스코드 기반의 화이트박스(White-Box) 시험을 할 수 있는 "실시간 이동형 내장 소프트웨어 테스팅 기술"이 필요하다. 실시간 테스팅 기술이 효율적으로 적용되기 위해서는 시험검증 도구가 필요하며, 이 도구를 지원하는 구조가 있어야 한다. 또한 실시간 이동형 내장 소프트웨어 시험검증 도구는 타겟시스템 기반의 테스팅이 가능해야 하고 실시간 기능을 제공할 수 있어야 한다.

그러나 실시간 이동형 내장 소프트웨어 테스팅을 위한 도구가 일부 제시되고는 있지만, 이 도구를 지원하는 구조가 잘 정립되어 있지 않고, 늘어나는 실시간 이동형 내장 소프트웨어 시험의 수요에 비해, 실질적으로 실시간 이동형 내장 소프트웨어 테스팅이 잘 이루어지지 않

· 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음
† 학생회원 : 숭실대학교 컴퓨터학과
hava67@selab.ssu.ac.kr
†† 정 회 원 : 숭실대학교 컴퓨터학부 교수
nylee@computing.ssu.ac.kr
††† 종신회원 : 숭실대학교 컴퓨터학부 교수
syrhew@comp.ssu.ac.kr
논문접수 : 2005년 3월 22일
심사완료 : 2006년 2월 20일

고 있다. 이유는 다음과 같다. 첫 번째, 임베디드는 플랫폼에 의존적이다. 기존의 다른 소프트웨어에 비해 임베디드 소프트웨어는 플랫폼, 즉 하드웨어 환경에 많은 영향을 받고 있다. 두 번째, 임베디드는 개발환경과 테스팅 환경이 다른 교차개발환경을 갖는다. 임베디드 테스팅 환경은 소프트웨어가 탑재된 플랫폼에서 이루어져야 하기 때문에, 일반적인 소프트웨어 테스팅 방법으로 임베디드 소프트웨어를 시험 하기에는 매우 어렵다.

그러므로 개발된 임베디드 소프트웨어를 시험검증하기 위한 시험 도구를 지원하는 구조가 필요하다.

본 논문의 구성은 2장에서 관련 연구로서 임베디드 소프트웨어 시험 도구 및 타 논문에서 제시한 임베디드 소프트웨어 구조를 소개한다. 3장에서는 실시간 이동형 내장 소프트웨어 시험 도구의 개발환경과 기능적 요구사항을 식별하고 시스템을 설계한다. 4장에서는 실시간 이동형 내장 소프트웨어 시험 도구를 지원하기 위한 구조를 설계한다. 기존의 MVC 모델 구조를 기반으로 추상화 시험 구조를 확장하며, 실시간 이동형 내장 소프트웨어 시험 구조를 설계하기 위해 구체화된 구조를 제안한다. 5장에서는 제안한 구조를 평가하며, 마지막으로 6장에서는 본 논문의 결론을 맺는다.

2. 관련연구

2.1 임베디드 소프트웨어 시험 도구[6]

임베디드 소프트웨어를 시험 하기 위한 시험 도구는 소프트웨어의 높은 품질을 보장하기 위하여 계획 및 제어, 명세, 초기 시험 파일, 시험 실행 및 평가와 같은 시험 활동을 지원할 수 있도록 자동화 되어야 하며, 시험 활동을 반복하여 실행할 수 있는 시험 프로세스를 지원해야 하고, 시험 활동의 모든 단계에 적용이 가능해야 한다.

시험 도구의 분류는 계획 및 제어(Planning and control) 단계, 준비(Preparation) 단계, 명세(Specification) 단계, 실행(Execution) 단계, 완료(Completion) 단계로 이루어진다.

계획 및 제어(Planning and control)단계는 시험과 결함을 관리하기 위한 도구로 사용되며 테스팅을 위하여 특별히 개발하지는 않는다. 이 단계에서는 결함(Defect)관리, 시험(Test)관리, 형상(Configuration)관리, 스케줄(Scheduling)과 모니터링(Monitoring)을 위한 도구들이 포함된다.

준비(Preparation)단계는 도구분석기(Case tool analyzer)와 복잡도분석기(Complexity analyzer)가 포함된다. 도구분석기는 일관성을 점검할 수 있어야 하고, 복잡도분석기는 소프트웨어의 복잡도에 대한 내용을 표시한다.

명세(Specification) 단계는 시험케이스생성기(Test case generator)가 포함된다. 시스템의 상태를 점검하기 위한 시험 케이스를 생성한다.

실행(Execution) 단계는 도구의 많은 타입들이 사용이 가능하다. 이 단계에서는 시험데이타생성기(Test data generator), 레코드 및 재생(Record and playback)도구, 로드 및 스트레스 시험(Load and stress test)도구, 시뮬레이터(Simulator), 스터브 및 드라이버(Stubs and drivers), 디버거(Debugger), 정적소스코드

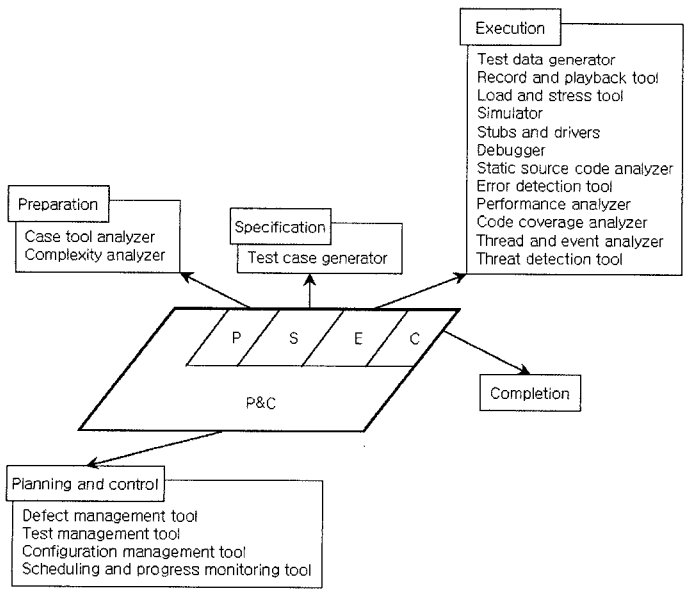


그림 1 라이프사이클 모델의 5단계

분석기(Static source code analyzer), 에러탐지(Error detection)도구, 성능분석기(Performance analyzer), 코드커버리지분석기(Code coverage analyzer), 쓰레드 및 이벤트분석기(Thread and event analyzer), 위험탐지(Threat detection)도구 등이 포함된다.

2.2 RK. White의 임베디드 소프트웨어 구조[7]

일반적으로 레거시 임베디드 애플리케이션들은 도메인 기반의 분할 방식을 사용하지 않고 물리적 분할 기법을 사용하여 애플리케이션들을 분할하고 있기 때문에 많은 문제점들이 발생하게 된다.

물리적으로 분리되는 소프트웨어 컴포넌트들은 완전한 공백상태에서만 개발이 가능하다. 개별적인 컴포넌트들은 어느 정도의 변경을 고려하여 변경으로부터 분리되도록 개발되어야 하지만, 물리적 분할에 의해서 독립적으로 개발되는 소프트웨어들은 시스템에 대한 고려없이 개발되기 때문에 시스템 통합시에 많은 문제들이 발생하게 된다.

또한 개별적인 기능들로 분리하기 위하여 할당하게 되는 기능들은 유사한 기능을 가지고 있는 코드를 포함하게 되기 때문에 소프트웨어 중복문제를 야기 시켜 소프트웨어의 개발비용과 유지보수 비용을 증가시키게 된다.

임베디드 소프트웨어의 새로운 요구사항들은 소프트웨어 시스템의 설계상의 잘못과 상관없이 발생하게되며, 레거시 임베디드 소프트웨어 시스템은 통합단계까지 많은 논리적인 의존 문제들을 야기시킨다. 이것은 소프트웨어의 변경이 전체 애플리케이션이 가지고 있는 결과를 결정하는 것을 매우 어렵게 만든다. 따라서 임베디드 소프트웨어 구조는 이러한 임베디드 시스템들의 잠재적인 문제점들을 해결할 수 있도록 설계되어야 한다.

2.3 N. Medvidovic의 소프트웨어 구조 및 임베디드 시스템[8]

소프트웨어 구조스타일(architectural styles)은 시스템에 적용 결과 시스템에서 되풀이하여 발생한 패턴들이다. 관련 구조는 스타일에 관련되어 있는 이슈이며 애플리케이션 또는 문제 도메인에 대하여 시스템에 적용할 수 있어야 한다.

임베디드 시스템에서 효율은 CPU의 최소 사용, 메모리, 배터리 파워, 네트워크 대역폭 등의 환경에 따른다. 임베디드 소프트웨어 시스템은 시뮬레이션 환경에서 개발되고 시험 된다. 타겟 환경은 시스템이 작동하는 동안 자주 변하기 때문에 호스트의 소프트웨어 컴포넌트 분산에 직접적으로 영향을 미친다. 따라서, 소프트웨어 설치와 소프트웨어 시스템의 업데이트를 도울 수 있는 배치 지원(deployment support)을 위한 요구가 늘어나고 있다. 임베디드 시스템은 배치에 앞서 자신의 기능들을 포함하여 타겟 환경을 검사하게 되므로 이 기능들을 애플리케이션의 구현 하부조직으로부터 제공받기 때문에, 이들 기능들을 호스트에 추가로 제공해야 한다.

동적 적응성(dynamic adaptability)은 시스템 구조의 변경이 가능하게 한다. 이들 변경은 구조 레벨에서 구조 모델과 실행중인 시스템 사이에 확실한 일관성을 유지할 수 있도록 하고 획득하는 것을 필요로 한다.

2.4 Lu Yan의 P2P와 Ad Hoc 오버레이 네트워크 애플리케이션을 위한 통합 구조[9]

P2P(Peer-to-peer) 시스템은 자동적이고, 분산방식 오버레이 네트워크이며, 참여 노드들이 자원을 제공하고 서비스를 제공하기 위하여 협력한다. Mobile ad hoc network(MANET)는 무선연결로 모바일 호스트(mobile hosts)를 연결하는 자동시스템이다.

P2P와 MANET를 위한 통합 구조인 MIN 구조를 제시한다. MIN 구조는 연결-레벨(link-level) MANET로 애플리케이션 레벨 P2P 오버레이 위에 만들어진다. 그러나 구조는 실행 환경에 구체적이지 않다. MIN 구조는 애플리케이션 계층(application layer)이라고 하는 추상 계층(abstract layer)을 제공한다. 또한 MIN 구조는 연결 결과에 적당한 애플리케이션 레벨 연결을 허용하는 논리적 연결, 즉 유선연결과 무선연결 또는 호스트들 사이의 가상연결을 제공한다.

2.5 N. TW. Pearce의 실시간 임베디드 시스템의 Simulation-Driven 구조[10]

시뮬레이션은 추상의 여러 가지 레벨에서 사용될 수 있는 컴포넌트 지향의 구조를 규정한다. 구조는 대규모 시스템의 컴포넌트로서 상호 운용에 시뮬레이션들을 허용하고, 시뮬레이션 산출물들이 타겟 시스템 컴포넌트와 쉽게 결합되도록 허용하고, 작업 제품들의 재사용을 촉진하며, COTS 컴포넌트의 포함을 허용하여야 한다.

3. 실시간 이동형 내장 소프트웨어 시험 도구[11]

본 논문에서 제안하는 실시간 이동형 내장 소프트웨어 시험 도구를 개발하기 위한 적용기술은 GNU의 GDB, GCC, LTT 등의 기술이 사용되며 이 중에서 알고리즘 부분만을 선택하여 사용할 수 있고, GNU에서 사용할 수 없거나 지원되지 않는 기술은 자체기술로 개발하여 사용이 가능하다. 개발환경은 호스트와 타겟이 분리되어 있으며, 호스트에서는 개발도구와 시뮬레이션 환경이 갖추어져 있고, 타겟에서는 임베디드용 타겟보드인 하드웨어와 임베디드 운영체제, 애플리케이션을 지원하는 드라이버(Driver)와 라이브러리(Library) 등으로 구성된다.

3.1 기능적 요구사항

실시간 이동형 내장 소프트웨어의 생산성을 높이고

신뢰성 있는 소프트웨어를 개발하기 위해서는 개발단계에서 생성되는 소스코드 기반의 화이트박스(White-Box) 테스팅을 원칙으로 하기 때문에 소스코드에 대한 분석모듈을 필요로 한다. 그리고 소스코드에 대한 분석 모듈을 위한 기능적 요구사항은 크게 다음의 8가지로 분류된다. 첫째, 소스코드 문법 파싱, 둘째, 소스코드 삽입기술, 셋째, 소프트웨어의 소스코드를 실시간으로 분석하기 위한 Testing Feature(Memory profiler, Code coverage profiler, Trace profiler, Performance profiler), 넷째, 에이전트(Host, Target), 다섯째, 자료저장소(Repository), 여섯째, 데이타 분석(Data analysis), 일곱째, 시험 슈트(Test suite), 여덟째, 보고서 생성(Report generation) 으로 분류된다.

이 중에서 시험 도구의 핵심 부분인 실시간 Testing Feature는 표 1과 같다.

실행 추적 프로파일러(Trace Profiler)는 타겟(Target) 프로그램을 실행하여 실시간 추적(Realtime tracing)을 생성한다. UML 순차도(Sequence Diagram)을 통해 기능(Function)간의 상호작용을 보여줌으로써 타겟 프로그램의 동적인 행위(제어흐름정보, 예외상황, 객체의 라이프사이클)를 이해하는데 많은 도움을 준다. 이 프로파일러는 추적(Trace)생성기, 뷰어(Viewer) 등을 포함한다.

메모리 프로파일러(Memory Profiler)는 실행시 메모리 관리에 대한 디버그(debug)를 하기 어렵고 중요한 정보가 되는 에러(Errors)와 메모리 누수(Leak)를 탐지하는 기능을 제공한다. 이 프로파일러는 FFM(Freeing Freed Memory), FUM(Freeing Unallocated Memory), MAF(Memory Allocation Failed), ABWL(Late Detect Array Bounds Write), FMWL(Late Detect Freed Memory Write), MIU(Memory In Use), MLK(Memory Leak), MPK(Potential Memory Leak), CD(Core Dump), FIU(File In Use) 등을 포함한다.

성능 프로파일러(Performance Profiler)는 애플리케이션의 실제 성능의 기준 및 애플리케이션이 제공할 수 있는 성능을 평가한다. 부하의 변화에 따른 가능한 병목(Bottleneck)현상과 문제를 해결하는 방법(시스템 변수의 튜닝, 소프트웨어 변경 또는 하드웨어 업그레이드)에 관한 유용한 정보를 제공해 준다. 이 프로파일러는 기능의 호출 횟수(Function Calls), 실행시간(Function Time), 최소 실행시간(Function Min-Time), 최대 실행시간(Function Max-Time), 전체 시험 시간에서 기능이 차지하는 실행시간(F Time(% of root)) 등을 포함한다.

코드 커버리지 프로파일러(Code Coverage Profiler)는 시험 수행 도중 소스 코드의 함수 등의 객체가 얼마나 실행됐는지 혹은 실행되지 않았는지를 보여준다. 이 프로파일러는 기능 커버리지(Function Coverage), 호출 커버리지(Call Coverage), 블록 커버리지(Block Coverage), 조건 커버리지(Condition Coverage), 커버리지 결과 보고(Coverage Result Report Viewer) 등을 포함한다[11,12]. 기능 커버리지(Function Coverage)는 시험 도중 함수가 얼마나 실행되었는지 혹은 실행되지 않았는지를 시험하는 기능이다. 호출 커버리지(Call Coverage)는 한 함수가 다른 함수를 얼마나 호출했는지를 시험하는 기능이다. 블록 커버리지(Block Coverage)는 블록이 얼마나 실행되었는지를 시험 하는 기능이다. 조건 커버리지(Condition Coverage)는 Boolean을 평가하는 표현식을 시험한다.

3.2 시스템 설계

3.2에서 제시한 실시간 이동형 내장 소프트웨어 시험 도구의 기능적 요구사항을 만족하는 시스템은 소스코드 삽입(Source Code Instrumentation), 호스트-타겟시스템 간의 통신채널(Communication Channel)을 지원하여 실시간 이동형 내장 소프트웨어 시험 도구 개발에 있어서 필수적인, 매우 유연하고 확장성 높은 타겟시스템 기반의 실시간 분석 시스템 구축을 가능하게 한다. 본 기술은 그림 2에서 처럼 그래픽 사용자인터페이스(GUI), 시험케이스생성기(TestCase Generator), 호스트 시험 실행(Host TestExecution), 호스트에이전트(Host Agent), 타겟에이전트(Target Agent), 타겟 시험 실행(Target TestExecution), 자료저장소(Data Repository) 등으로 구성된다.

표 1 실시간 Testing Feature

실시간 Testing Feature	기능
트레이스 프로파일러 (Trace Profiler)	타겟 기반의 실행추적(Runtime Tracing) 결과를 UML의 순차도(Sequence Diagram) 형태로 나타낸다.
메모리 프로파일러 (Memory Profiler)	Heap 공간에 설정된 메모리 누수를 찾아낸다.
성능 프로파일러 (Performance Profiler)	Function 및 Method의 수행을 나타내어 애플리케이션의 병목지점을 찾아낸다.
코드 커버리지 프로파일러 (Code Coverage Profiler)	코드 수행에 대한 적용범위를 분석한다.

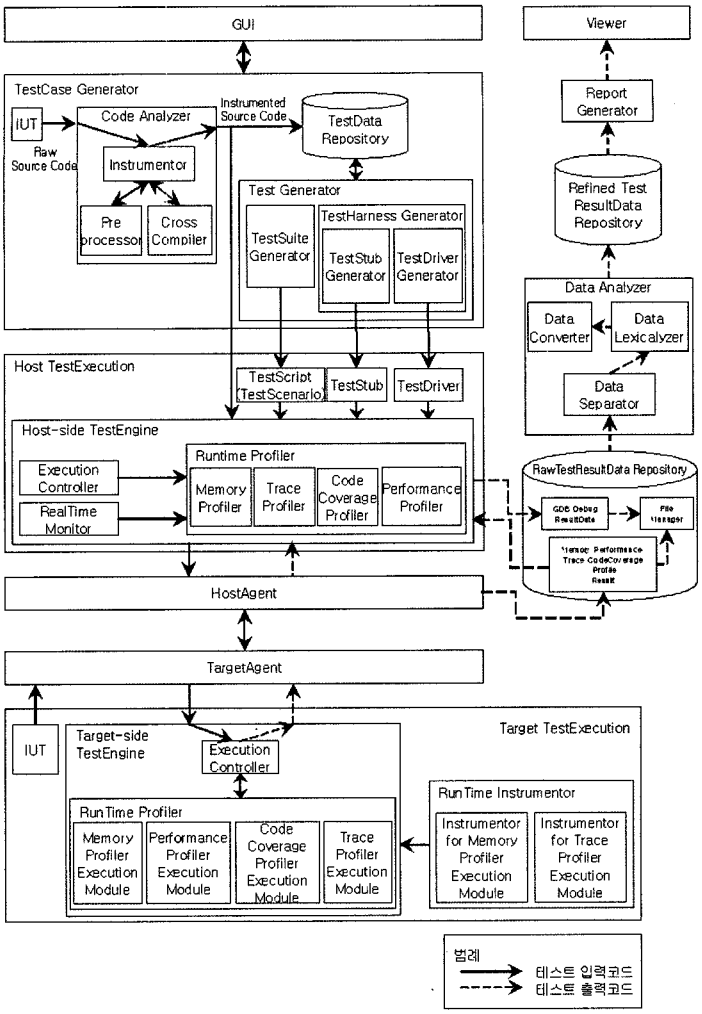


그림 2 실시간 이동형 내장 소프트웨어 시험 시스템 설계

4. 실시간 이동형 내장 소프트웨어 시험 구조 설계

본 논문에서 제안하는 실시간 이동형 내장 소프트웨어 시험 도구를 위한 구조는 적합한 구조를 설계하기 위하여 가능한 구조의 모든 스타일을 고려하며, 임베디드 시스템의 특수한 개발환경인 교차개발환경에 따른 상황을 고려하고, 호스트시스템과 타겟시스템 사이의 정보교환을 위한 네트워크를 고려하여 설계한다.

4장에서는 실시간 이동형 내장 소프트웨어 시험 구조를 구성하는 방법에 대하여 계층(layer) 스타일을 적용한다. 시스템을 계층으로 나누었을 때의 장점은 다음과 같다. 첫째, 다른 계층들에 대해 많이 알지 못해도 단일 계층을 응집된 전체로 이해할 수 있다. 둘째, 동일한 기본 서비스의 대안 구현으로 계층을 대체할 수 있다. 셋째, 계층들 사이의 의존성을 최소화한다. 넷째, 계층은 표준화를 위한 훌륭한 역할분담의 장치를 제공한다. 다섯째, 계층을 한번 설계해 놓으면, 여러 상위 수준의 서비스에 사용할 수 있다.

따라서 본 논문에서는 유사한 기능들의 집합을 식별하기 위하여 물리적 분할 기법을 사용하지 않고 도메인 기반의 분할 기법을 사용하여 기능을 분할한다. 이미 검증된 MVC 모델을 확장하여 임베디드에 적용하고 구조 설계 기법을 최적화하여, 시험 수행시 구조의 유연성을 보장하고 효율적인 시험을 수행하기 위하여 플랫폼에 독립적인 소프트웨어 구조를 설계한다.

4.1 개념적 구조 설계

그림 3(a)처럼 MVC 모델 구조를 실시간 임베디드 환경에 맞게 세분화하면 다음과 같다. 먼저 모델(Model)

은 데이타베이스 계층과 데이타접근 계층으로 나누어지고, 컨트롤러(Controller)는 비즈니스도메인 계층으로 나누어지며, 뷰(View)는 프리젠테이션 계층과 접근 계층으로 나누어진다[13-15].

프리젠테이션 계층, 접근 계층, 비즈니스도메인 계층, 데이타접근 계층, 데이타베이스 계층에 대하여 계층 스타일(Layered Style)을 사용한다. 또한 각 모듈에서 데이타를 사용하는 것을 추적하고 영구적인 데이타의 변화를 추적하기 위해서 공유 데이타 스타일(Shared-Data Style)을 사용한다[15].

프리젠테이션 계층(Presentation Layer)은 모델의 데이타를 통하여 사용자의 정보를 입력받고 분석된 결과를 사용자에게 보여준다. 또한 사용자의 이벤트를 받는 장치 역할을 하며, 데이타베이스에서 시험된 결과를 그래프나 표로 변경하여 프리젠테이션 장치에서 볼 수 있도록 해주는 기능을 갖는다.

접근 계층(Access Layer)은 프리젠테이션 계층과 시스템과의 연결을 담당한다. 예를 들어, 호스트와 타겟 사이의 통신채널, 호스트 에이전트와 타겟 에이전트를 통하여 데이타를 받을 수 있도록 네트워크를 연결한다.

비즈니스도메인 계층(Business Domain Layer)은 비즈니스 로직(Business logic)을 가지고 있으며, 실시간 이동형 내장 시스템에서 실제로 작동하는 대부분의 기능을 갖는다. 여러 가지 시스템을 묶어 애플리케이션을 만들며, 뷰로부터 사용자의 요구사항을 받아 시스템을 작동시킨다.

데이타접근 계층(Data Access Layer)은 호스트와 타겟의 시험 전 데이타와 시험 후의 결과 데이타를 지원한다. 물리적 데이타베이스와 파일시스템에 관련된 데이타를 저장할 수 있도록 지원하며, 호스트 에이전트와 타겟 에이전트를 통하여 전송된 데이타를 분석하여 시험 전 데이타와 시험 후의 결과 데이타를 식별하여 데이타 처리를 수행하고 변경되는 데이타를 감시한다.

데이타베이스 계층(Database Layer)은 물리적 데이타가 있는 계층이다. 애플리케이션의 시험 데이타의 정보를 저장하고 임베디드 시스템을 지원 가능하도록 한다.

4.2 상세 구조 설계

그림 3(b)처럼 프리젠테이션 장치, 사용자 인터페이스, 타겟 에이전트 계층, 호스트 에이전트 계층, 타겟 실행 계층, 호스트 실행 계층, 호스트 실행 지원 계층, 데이타베이스 CRUD, 자료저장소(Repository) 등으로 구성된다.

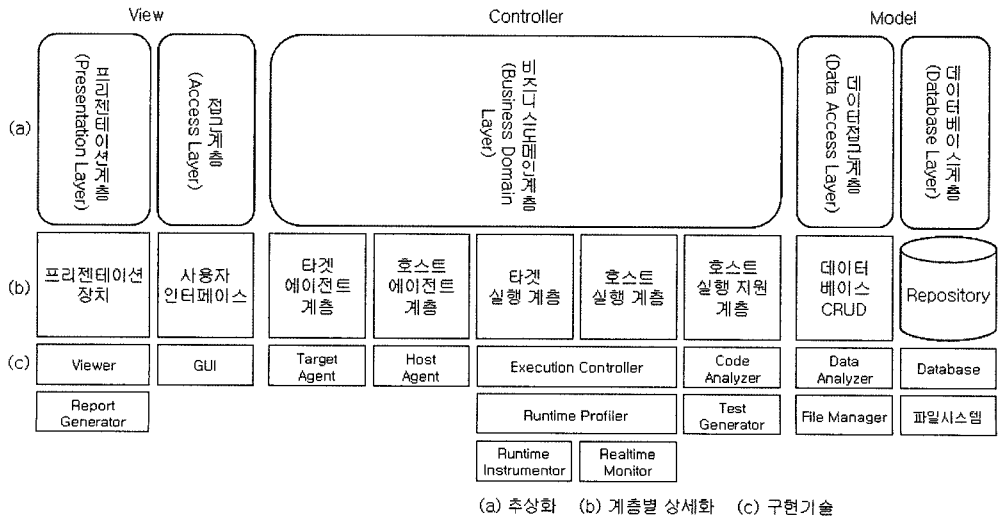
프리젠테이션 장치는 사용자와 시스템간의 대화를 위한 장치이다. 이 계층은 모델의 데이타 정보를 입력받고 분석된 결과를 사용자에게 보여준다. 뷰어(Viewer)와 결과 생성(Report Generator) 등으로 구성된다.

사용자 인터페이스는 프리젠테이션 계층과 시스템과의 연결을 담당한다. 이 계층은 GUI로 구성된다.

호스트 에이전트 계층(Host Agent Layer)은 타겟 시스템과 정보를 교환하기 위한 네트워크를 설정하여 타겟시스템과 통신을 한다.

타겟 에이전트 계층(Target Agent Layer)은 호스트시스템과 정보를 교환하기 위한 네트워크를 설정하여 호스트시스템과 통신을 한다.

호스트 실행 계층(Host Execution Layer)은 사용자가 정한 기준에 따라 충분한 시험이 수행될 때까지



(a) 추상화 (b) 계층별 상세화 (c) 구현기술

그림 3 MVC 모델을 기반으로 한 실시간 이동형 내장 소프트웨어 시험 구조

시험데이타(Test Data)를 생성하여 시험 수행을 하는 계층이다. 시험스크립트(Test Script)와 시험데이타가 준비되면 실제 시험을 수행하기 위해 시험대상프로그램(Implementation Under Test, IUT)을 실행하고, 시험의 결과를 보고하기 위해 시험 수행 과정을 모니터링한다. 또한 타겟시스템(Target System)에서 실행될 때의 추가조건정보를 추가하고 타겟시스템에서 전송해온 정보를 사용자가 원하는 형태로 변환한다. 이 계층은 실행제어기(Execution Controller), 실행프로파일러(Runtime Profiler), 실시간모니터(Realtime Monitor) 등으로 구성된다.

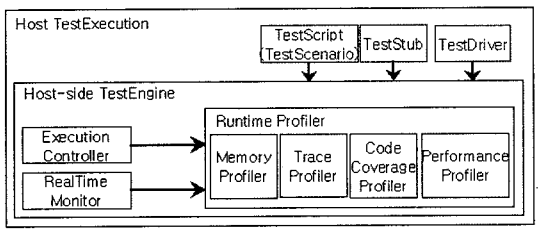


그림 4 호스트 실행 계층

타겟 실행 계층(Target Execution Layer)은 호스트 실행 계층에서 생성된 시험데이타를 호스트 에이전트와 타겟 에이전트를 통하여 넘겨받아 실제 시험을 수행하고, 실행 중 사용자가 원하는 정보를 추출해낸다. 이 계층은 실행제어기(Execution Controller), 실행프로파일러(Runtime Profiler), 실행삽입기(Runtime Instrumentor) 등으로 구성된다.

호스트 실행 지원 계층(Host Execution Support Layer)은 시험에 요구되는 정보를 생성해 내기 위하여 시험대상프로그램을 분석하고, 시험데이타와 시험시나리오(Test Scenario)를 생성한다. 이 계층은 코드분석기(Code Analyzer)와 시험생성기(Test Generator) 등으로 구성된다.

데이타베이스 CRUD는 데이타 분석과, 파일관리 기법을 사용하여 데이타베이스 또는 파일시스템에 데이타를 생성, 수정, 삭제 및 검색을 수행하고, 애플리케이션의 데이타를 보호한다. 이 계층은 시험이 수행되기 전에 생성해낸 데이타와 타겟에서 실제 시험을 수행한 후의 결과 데이타를 관리하기 위해 호스트에서 관리한다. 이 계층은 데이타분석기(Data Analyzer)와 파일관리자(File Manager)등으로 구성된다.

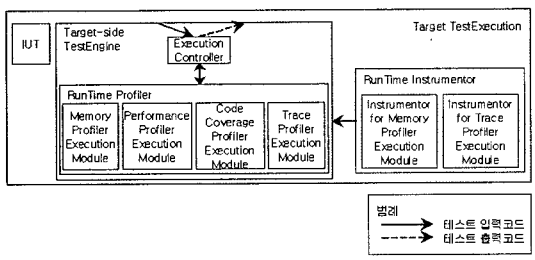


그림 5 타겟 실행 계층

자료저장소(Repository)는 시스템에서 관리하여야 하는 물리적 데이타베이스이다. 이 계층역시 호스트에서 관리하며 데이타베이스, 파일시스템 등으로 구성된다.

4.3 구조의 적용

실시간 이동형 내장 소프트웨어 시험 도구의 구조를 구성하고 있는 구성요소는 그림 3(c)처럼 뷰어(Viewer), 보고서생성기(Report Generator), 타겟에이전트(Target Agent), 호스트에이전트(Host Agent), 실행제어기(Execution Controller), 실행프로파일러(Runtime Profiler), 실행삽입기(Runtime Instrumentor), 실시간모니터(Realtime Monitor), 코드분석기(Code Analyzer), 시험생성기(Test Generator), 데이타분석기(Data Analyzer) 등으로 구성된다.

에이전트 실행 계층은 그림 6에서 처럼 실시간 이동형 내장 소프트웨어의 성능을 효율적으로 측정하기 위하여 타겟 시스템에 성능 측정 에이전트라는 작업(task)을 수행시켜 호스트에서 결과를 받아 볼 수 있는 클라이언트-서버 구조를 정의한다. 성능 측정 에이전트는 호스트와 타겟 사이의 TCP/IP 소켓 통신을 통해 시험대상소스코드와 시험 결과를 전달한다. 모든 테스팅 결과는 호스트쪽의 에이전트 모듈에 의해 관리되며, 저장소(Repository)에 저장되는 데이타 형태로 변환하기 위해 데이타분석기(Data Analyzer)에게 전달된다. 타겟 에이전트는 시험대상소스코드에 대해 실행되는 실제 프로파일 모듈을 구동시키기 위한 성능 측정 스터브(stub)를 가지며, 시리얼 포트나 이더넷을 통해 요청하는 정지점(Breakpoint)관리, 레지스터와 메모리의 조회/수정, 스택 프레임 조회, 이벤트 관리, 예외핸들러(Exception handler) 관리 등에 대한 처리를 하고 호스트로 응답하는 기능을 담당한다. 호스트에이전트와 타겟에이전트 간에 이벤트와 메시지를 전달해주는 역할을 하는 인터페이스는 퍼사드 패턴(Façade Pattern)을 사용한다. 퍼사드 패턴은 단순한 인터페이스를 제공하여 호스트에이전트와 타겟에이전트 사이에 의존성(dependency)을 줄여주며, 계층간의 결합도(coupling)를 낮추어 준다[16].

실시간 이동형 내장 시스템의 경우 호스트에이전트와 타겟에이전트 사이에는 수많은 트랜잭션이 발생하게 된다. 트랜잭션이 발생할 때마다 메소드를 호출하게 되면 시스템이 부하가 많이 걸리게 된다. 그러나 퍼사드 패턴을 사용하여 단순한 형태로 통합하여 인터페이스를 제공하게 되면 메시지 호출과 순서를 하나의 트랜잭션 단

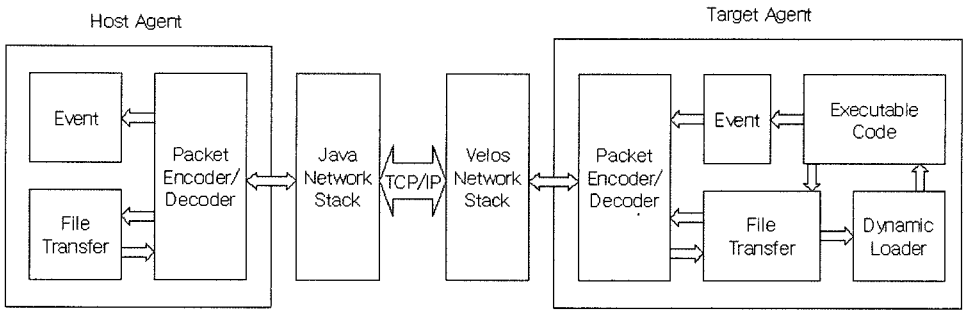


그림 6 에이전트 전체 구조도

위로 묶어서 관리할 수 있게 된다. 그러므로 호스트에이전트와 타겟에이전트 사이의 메시지 호출 횟수를 줄여 효율적이고, 병목 현상을 줄여준다.

자료저장소(Repository) 계층은 그림 7에서 처럼 임베디드 소프트웨어 시험 수행 과정에서 필요한 소스코드삽입정보를 저장하는 기능을 갖는다. 시험데이타 저장소(Testdata Repository)와 호스트측 시험 엔진을 통해 분석된 결과를 저장하는 로시험데이타 저장소(Raw Testdata Repository), 통합된 로시험데이타 저장소에 저장된 데이타를 데이타분석기에 의해 시험특징(Testing Feature)에 따라 분리된 정보를 저장하고 결과 생성을 위해 참조파일(Reference File)을 저장하는 시험결과데이타저장소(Refined Test Result Data Repository)로 구성된다.

데이타분석기(Data Analyzer)는 그림 7에서 처럼 타겟에서 실행된 소스코드에 대한 시험 결과를 분석하고 저장소에 저장되는 데이타 형태로 결과를 변경하기 위한 모듈이다. 데이타분석기는 다음과 같이 플래그(Flag), 입력결과데이타(InputResultData), 데이타변환기(Data Converter), 어휘분석기(Data Lexical Analyzer), 데이타분리기(Data Separator) 등으로 구성된다. 플래그는 시험결과데이타가 4가지의 프로파일(profile) 중에 어느 프로파일 결과 인지를 어휘분석기에게 알려주는 정보를 나타낸다. 입력결과데이타는 실제 시험결과를 읽어 들여서 읽어 들인 데이타를 어휘분석기에게 전달한다. 데이타변환기는 프로파일에 따라 토큰(tokenizing)된 결과를 저장소에 저장하기 위한 데이타 형태로 변환한다. 어휘분석기는 데이타분리기에 의해 플래그에 따라 선택된 프로파일 종류를 선택하고 해당 의미에 맞는 토큰을 구별한다. 데이타분리기는 입력시험결과를 플래그에 따라 토큰을 분류하여 어휘분석기를 호출한다.

시험케이스생성기(Test Case Generator)는 그림 8에서 처럼 시험케이스생성기는 시험 하고자 하는 원시소스를 분석하여 시험 입력값과 그에 상응하는 예상 결과값을 자동으로 생성한다. 프로그램 시험 과정에서 가장

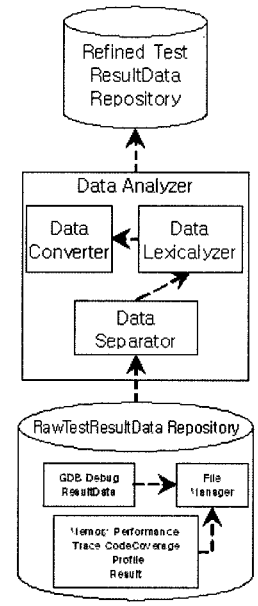


그림 7 자료저장소(Repository) 및 데이타분석기(Data Analyzer)

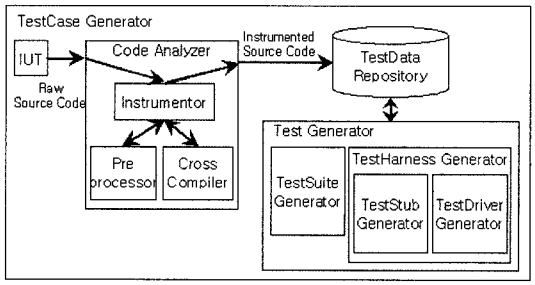


그림 8 시험케이스 생성기

중요한 것은 효과적인 시험케이스를 설계하는 것이다. 시험케이스 설계 작업이 중요한 이유는 모든 에러를 찾는 테스팅이 불가능하기 때문에 소프트웨어를 시험 할 때는 불완전한 테스팅을 할 수 밖에 없다. 따라서 시험

과정은 가능한 불완전성을 줄이는데 초점이 맞추어 진다. 시험케이스생성기는 시험하려고 하는 소프트웨어가 요구사항에 맞게 개발되었는지를 확인하기 위하여 시험 자동화를 통하여 시험데이타를 입력하고, 시험 예상값과 결과값이 일치하는지를 확인한다. 시험케이스생성기는 시험대상프로그램(IUT), 코드분석기(Code Analyzer), 시험생성기(Test Generator) 등으로 구성된다. 시험케이스는 시험케이스 이름, 시험하고자 하는 기능들의 항목(function, processes, services), 기능들에 적용될 입력들의 이름과 값의 리스트, 기능들의 수행한 결과로 얻어질 출력값들의 리스트 등으로 구성된다.

시험슈트생성기(Test Suite Generator)는 그림 8에서 처럼 사용자에 의해 생성된 시험케이스(Test Case)의 집합인 시험 슈트를 생성하는 모듈이다. 요구사항에 맞게 개발되었는지 확인하기 위하여 시험할 입력값과 예상 결과값을 정의한 시험케이스와 시험데이타를 생성한다. 사용자에 의해 각각의 함수에 대한 시험케이스를 추가하여 하나의 시험 슈트를 XML 문서형식과 소스코드로 저장한다.

시험대상프로그램(IUT)을 읽어오면, 코드분석기에 의해 소스코드를 기반으로 구문분석을 통하여 시험에 필요한 정보 즉, 변수정보와 함수정보, 변수 및 함수의 타입정보, 프로그램 제어흐름 정보를 추출하여 화면에 보여준다.

그림 9처럼 시험슈트를 생성하기 위해 시험슈트 이름을 설정하고, 변수정보와 함수정보, 시험케이스 정보를 선택하면 XML 문서형식으로 시험슈트를 생성하여 화면에 보여준다.

소스코드삽입기(Instrumentor)는 시험 대상 소스코드 내에 실시간 분석을 위한 4가지 테스팅 특징(Testing Feature)에 해당하는 특정코드를 삽입하는 기술이다. 이들 소스 내에 추가된 특정 코드는 기존 개발 환경에서 완벽하게 인식 가능한 형태로 실시간 분석 기능을 위해 식별자 및 시험 라이브러리(Testing Library) 사용을 추가시키거나 기존의 원시 라이브러리(Native Library) 코드를 시험 라이브러리 호출로 대체시킨다.

메모리프로파일링코드삽입기(Memory Profiling Code Instrumentor)는 내부적으로 메모리 할당과 해제 내역을 저장하는 리스트를 유지 관리하며, 기존의 메모리 할당, 해제함수를 호출하기 전에 리스트의 할당 해제 내역을 참고하여 문제가 발생할 수 있는 부분을 찾아낸다. 그림 10은 메모리프로파일러삽입기에 의하여 추출된 메모리프로파일러 실행화면이다.

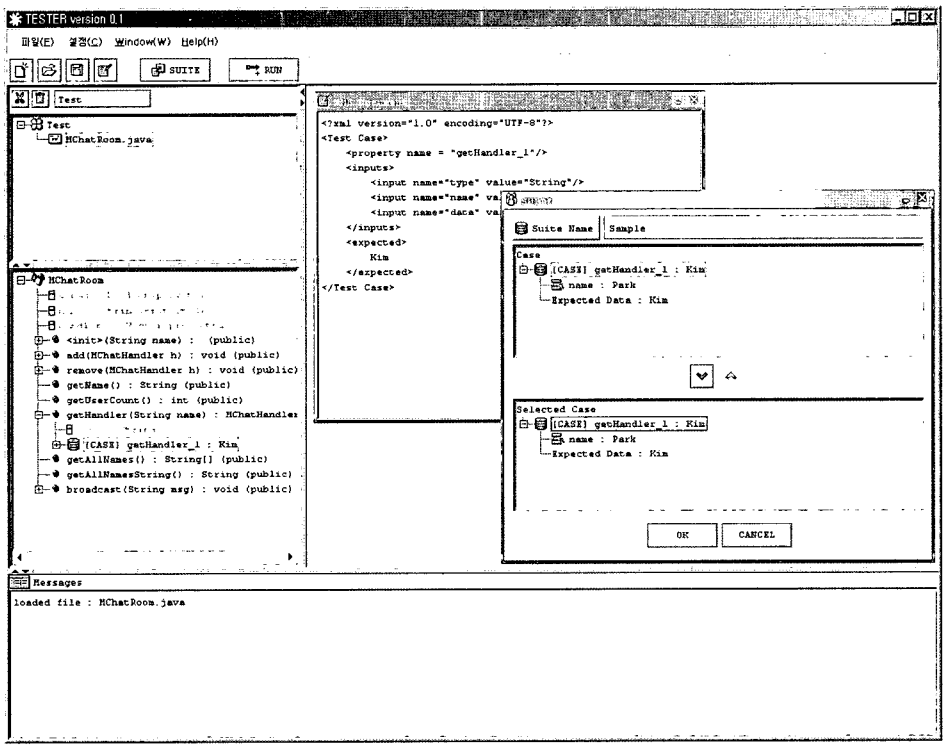


그림 9 시험슈트 생성 화면

성능프로파일링코드삽입기(Performance Profiling Code Instrumentor)는 현재 실행 중인 함수와 현재실행시간을 저장하고 있다가 프로그램이 종료되면 파일로 저장하는 기능을 갖는다. 이 부분은 임베디드 타겟(Target)시스템에서 호스트(Host)로 지속적인 이벤트를 전달해야 하기 때문에 동일한 함수원형을 가지는 새로운 함수를 만들어 함께 링크한다. 그림 11은 성능프로파일러 삽입기에 의하여 추출된 성능프로파일러 실행화면이다.

코드커버리지프로파일링코드삽입기(Code Coverage Profiling Code Instrumentor)는 프로파일링 설정에 따라 각 함수와 블록의 진입점과 종료점에 실행 이벤트를 기록하는 함수를 호출하는 코드를 삽입한다.

- function entry & exit point logger : 실행된 function 수 / 전체 function 수 <= 1
- call logger : 실행된 함수 call 수 / 전체 함수 call 수 <= 1
- block entry & exit point logger : 실행된 block 수 / 전체 block 수 <= 1

그림 12는 코드커버리지프로파일러삽입기에 의하여 추출된 코드커버리지프로파일러 실행화면이다.

트레이스프로파일링코드삽입기(Trace Profiling Code Instrumentor)는 각 함수의 진입점(entry point)과 종료점(exit point)에 현재실행시간을 기록하는 함수를 호출하는 코드를 삽입한다. 그림 13은 트레이스프로파일러삽입기에 의하여 추출된 트레이스프로파일러 실행화면이다.

실행프로파일러(Runtime Profiler)는 그림 5에서 처럼 소스코드 삽입기에 의해 원시코드에 삽입된 추가적인 코드들로부터 프로그램 내부에서 어떤 부분이 시간을 많이 사용하는지, 어떤 함수를 많이 호출하는지, 함수 사이에 상호 호출관계가 어떤지에 대한 정보 즉, 각 프로파일(Profile) 정보가 추출되고 시간적 순서에 따라 이벤트가 발생된다. 타겟 측에서는 이러한 저수준 프로파일 정

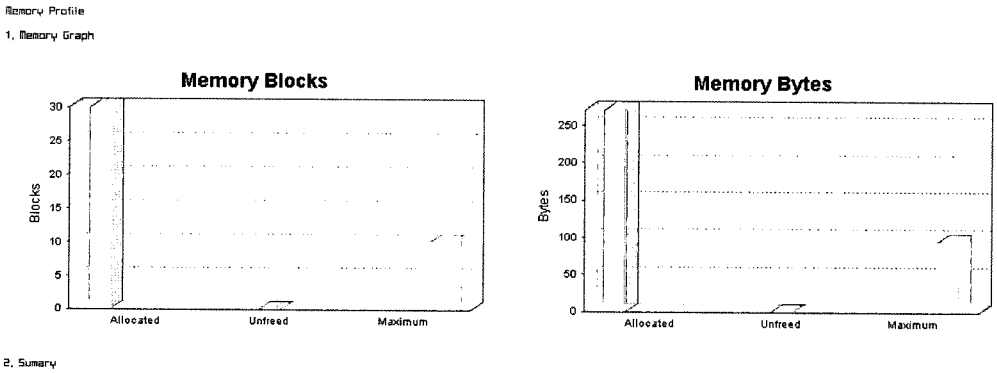


그림 10 메모리프로파일러 실행화면

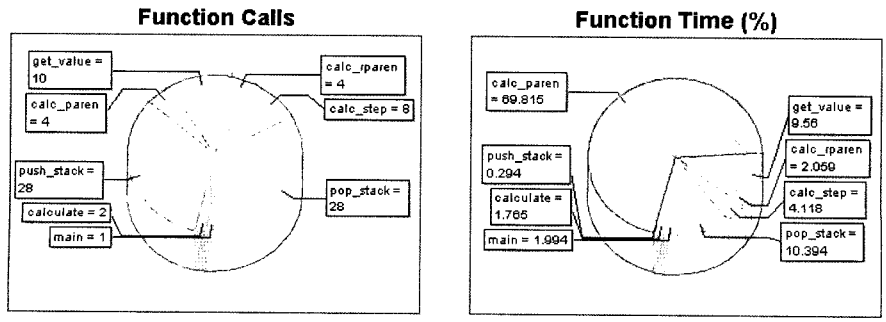


그림 11 성능프로파일러 실행화면

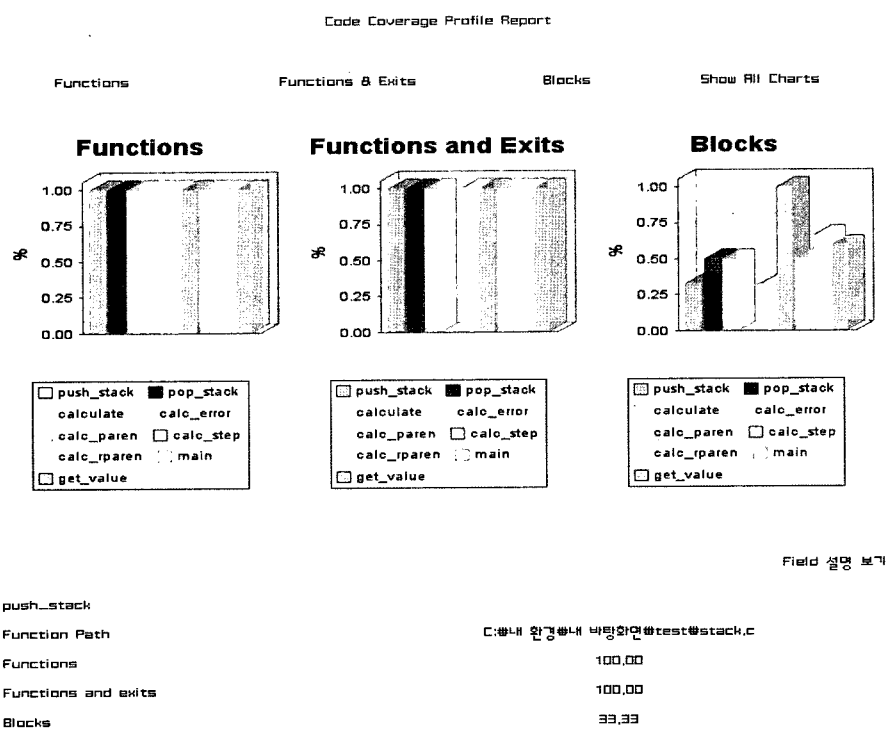


그림 12 코드커버리지프로파일러 실행화면

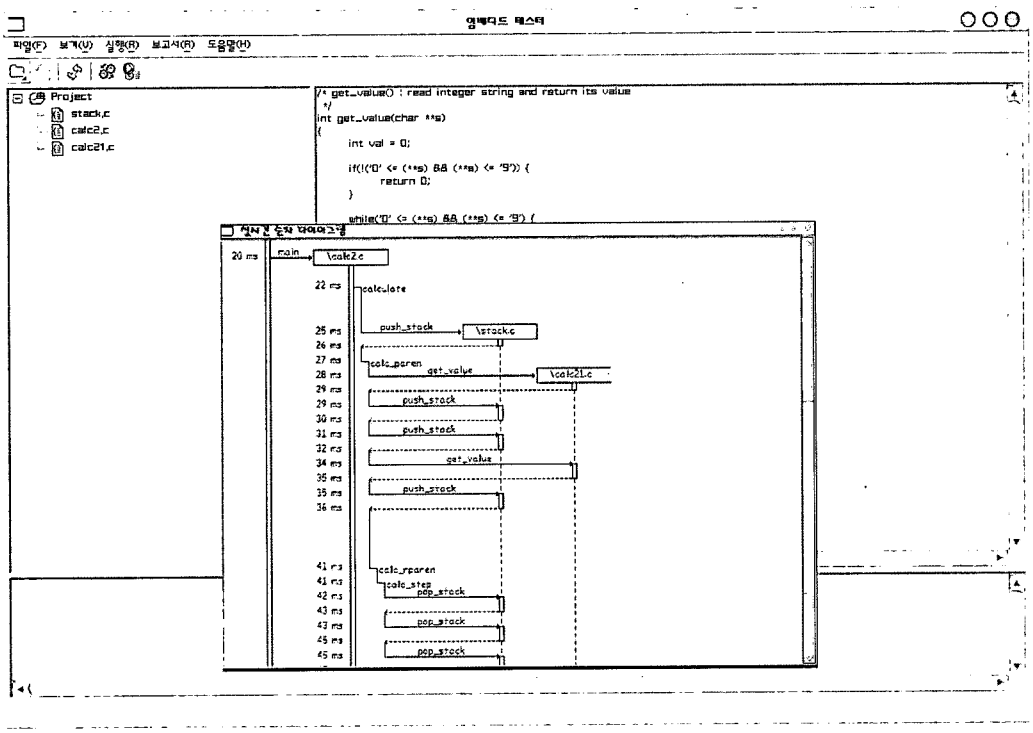


그림 13 트레이스프로파일러 실행화면

보들을 수집하고, 호스트 측으로 전송하기 적합한 형태로 변환하여 타겟에이전트(Target Agent)로 보낸다.

일반적인 네트워크 패킷 처리량에 비해서 실행 시간에 따라서 발생되는 이벤트의 수가 훨씬 많으므로 데이타를 가능한 적은 용량으로 압축하여 전송할 필요가 있다. 개발 과정에서 검증을 마친 후, 통합단계에서의 오류를 찾기 위한 실행프로파일러는 표 1과 같다.

실행프로파일러에서는 서로 다른 메시지들을 객체화하여 에이전트를 통하여 호스트에게 파라미터로 넘겨줄 수 있도록 하는 커맨드 패턴(Command Pattern)을 사용한다. 실시간 이동형 내장 시스템의 경우 타겟실행계층의 실행프로파일러에서 커맨드 패턴을 사용하게 되면 호스트로 부터 들어오는 서로 다른 메시지들을 객체화하여 에이전트를 통하여 호스트에게 파라미터로 넘겨줄 수 있게 된다. 장점은 타겟측에서 실행된 프로파일(profile) 정보에 대한 로그를 남겨 두면 시스템이 고장났을 때 원상태로 복구가 가능하다. 또한 커맨드 인터페이스를 확장하여 비즈니스 로직을 정의하면 프로파일링된 로그의 정보를 자료저장소(repository)에 저장해 둘 수 있게 된다. 시스템에 장애가 발생했을 때 타겟시스템에 저장된 로그를 읽어서 재실행할 수 있다[16].

5. 평 가

본 논문에서는 실시간 이동형 내장 소프트웨어 시험 도구와 도구를 지원하는 구조를 제안하였다. 본 논문에서 제시한 구조를 기반으로 4.3절에서 C언어 소스코드를 대상으로 한 이동형 내장 소프트웨어 시험에 적용하였다. 적용한 구조는 성능, 변경성, 유지보수성, 재사용성 등의 구조의 품질속성을 만족한다. 또한 관련연구에서 제시한대로 도메인 기반의 분할 기법을 사용하여 구조를 설계 하였기 때문에 개발 도메인의 외부 시스템이 변경되어도, 그에 따른 시스템의 변경이 줄어든다.

제안된 구조의 모델은 이미 검증된 MVC 모델을 확장하였으며, 각각의 시스템의 역할을 계층화시켰다. 또한 4장에서 제시한 실시간 이동형 내장 소프트웨어 시험 구조의 정의에 대하여 적용해야 하는 현재의 기술에 대한 부분을 만족시키기 위해 각 계층에 대한 기반 기술과 구현 기법을 제공하였다.

제안된 구조의 모델로 구축된 시험 시스템은 시스템을 관리 및 감독하기 위하여 호스트와 타겟 간의 실행을 제어하는 부분, 작업흐름 계층에 사용된 시스템을 실시간으로 모니터링 하는 부분, 타겟 기반의 실행추적(Runtime Tracing) 부분, 메모리 누수를 찾아내는 부분, 기능 수행에 대한 애플리케이션의 병목지점을 찾아내는 부분, 코드의 수행에 대한 적용범위를 분석하는 부분 등에 대한 관리 및 감독이 가능하다.

제안한 구조는 도메인 기반의 분할 기법을 사용하였기 때문에 계층간의 결합도를 낮추어주고, 다른 계층의 기능이 교체되거나 변경되더라도 그 영향을 줄여주며, 유지보수를 용이하게 해주며 플랫폼에 독립적으로 사용이 가능하다.

본 논문에서 제안한 실시간 이동형 내장 소프트웨어 시험 구조를 평가하기 위하여 품질속성 기반 구조 스타일(Attribute-Based Architectural Styles), Architecture Tradeoff Analysis Method(ATAM) 기법[17] 등에서 추출한 평가항목을 사용하고, 다른 논문에서 제시된 4개의 구조[7-10]와 비교 평가한 결과는 표 2와 같다.

6. 결 론

본 논문에서는 개념적 수준의 구조 계층뿐만 아니라, 제안된 구조를 구체적으로 제시하기 위해, 각 계층에 적용되는 실시간 임베디드 소프트웨어 시험 구현기법을 제시하였으며, 이들 기법을 최적화하여 호스트와 타겟 간의 상호작용, 실행 제어, 실시간 모니터링, 실행 프로파일 등에 효율적인 '실시간 이동형 내장 소프트웨어 시험 구조'를 제안하였다.

기존의 임베디드 시험 도구 및 관련 시험 논문들에서는 모바일 기반의 실시간 임베디드 소프트웨어 시험 도구에 적합한 구체적인 구조의 제시가 없었으며, 관련기술과 이들에 대한 상세한 지침이 없었다.

본 논문에서는 MVC 모델 구조를 확장하여 실시간 이동형 내장 소프트웨어 시험 도구를 지원하는 구조로 발전시키기 위해 추상화시키고 구체화시키는 절차적 방법에 의해서 실시간 이동형 내장 소프트웨어 시험 도구의 구조를 설계하는 기법을 제시하였다.

제안된 구조와 해당계층에 적절한 설계기법을 사용하여 시험검증시스템을 구축하면 실시간 이동형 내장 소프트웨어 시험에 많은 도움이 될 것으로 기대한다.

참 고 문 헌

[1] Desmond F. D'Souza and Alan Cameron Wills, Objects, Components, and Frameworks with UML, The Catalysis Approach, Addison-Wesley, 1999.

[2] Kazman, Software Architecture in Practice, 2nd Ed., Addison-Wesley, 2003.

[3] D. Garlan and M. Shaw, "An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering," Vol.1, World Scientific Publishing Company, pp. 1-39, 1993.

[4] Philippe Krutchen, "The 4+1 View Model of Architecture," IEEE Software, 12(6), pp. 42-50, 1995.

[5] M. Shaw and D. Garlan, Software Architecture:

표 2 타 구조와 제안한 구조 비교 평가

항목	RK. White	Lu Yan	N. Medvidovic	TW. Pearce	제안한 구조
구조 제안동기	성능	통합	구현	시뮬레이션 구조	시험, 구체적
제시 플랫폼	임베디드	P2P네트워크	임베디드,미들웨어	임베디드	모바일, 임베디드
추상 구조 계층(layer)	○	○	○	○	○
구체적 구조 계층 (layer)		○	○		○
재사용성 및 대체성	○			○	○
확장성 및 변경성	○				○
네트워크 연결		○	○		○
성능 및 신뢰도		○	○	○	○
유지보수성			○	○	○
계층간 낮은 결합도					○
도메인 분할 방법	○				○
외부 시스템 연동		○			○
트랜잭션 처리		○		○	○
구현 패턴	○				○
실시간 시스템 모니터링	○		○		○
실행 제어					○
실행 추적					○
메모리 분석			○		○
성능 분석					○
코드 커버리지 분석					○

Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[6] B. Broekman and E. Notenboom, Testing Embedded Software, p.209, Addison-Wesley, 2003.

[7] RK. White and U. Syyid, "Architecture Driven Software Design For Embedded Systems," UK Technology Innovation and growth Forum, 2004. http://whitepapers.zdnet.co.uk/0,39025945,60009152p-39000493q,00.htm

[8] N. Medvidovic, S. Malek, and M. Mikic-Rakic, "Software Architectures and Embedded Systems," SEES, pp. 65-71, 2003.

[9] L. Yan and K. Sere, "Towards an Integrated Architecture for Peer-to-Peer and Ad Hoc Overlay Network Applications," IEEE, 2004.

[10] Trevor W. Pearce, "Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems," RTSS, 2003.

[11] 산업자원부, "제조 산업을 위한 실시간 임베디드 S/W 테스팅 기술 및 시스템 개발- 위탁 연구 보고서", 2004.

[12] 김행곤, "객체 및 컴포넌트 기반 소프트웨어 공학", 도서출판 그린, 2002.

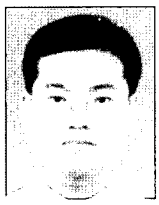
[13] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[14] 민현기, 김수동, "Effective Design Pattern and Enterprise Architecture Design Techniques in EJB Environment," 정보과학회논문지, 제30권, 제11호, pp. 1025-1036, 2003.

[15] P. Clements, L. Bass, and D. Garlan, Documenting Software Architectures, Addison-Wesley, 2002.

[16] E. Gamma, R. Helm and R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[17] P. Clements and R. Kazman, Evaluating Software Architectures, Addison-Wesley, 2001.



김 상 일

1998년 숭실대학교 소프트웨어공학과 공학사. 2001년 숭실대학교 정보과학대학원 공학석사. 2003년 숭실대학교 일반대학원 컴퓨터학과 박사수료. 관심분야는 소프트웨어 테스팅, 모바일/임베디드 소프트웨어 테스팅, 소프트웨어개발방법론, 소프트웨어 재사용, 디자인패턴, 리팩토링

이 남 용

1979년 숭실대학교 전자계산학과 공학사 1983년 고려대학교 경영대학원 경영학석사(MBA). 1993년 미국 미시시피주립대학교 대학원 경영학박사(MIS). 1979년~1983년 국군정보사 정보시스템분석장교(ROTC 17#). 1983년~1999년 한국국방연구원(KIDA), 국방정보체계연구소(KIDIS), 국방과학연구소(ADD). 1999년~현재 숭실대학교 정보과학대학 컴퓨터학부 교수. 관심분야는 소프트웨어 테스팅, EC/MIS/ISP, S/W Metrics, EA

류 성 열

1997년 2월 아주대학교 컴퓨터학부(공학박사). 1997년 3월~1998년 3월 George Mason University 교환교수. 1981년 3월~현재 숭실대학교 정보과학대학 컴퓨터학부 교수. 1998년 3월~2001년 2월 숭실대학교 정보과학대학원 원장. 1998년 3월~2004년 7월 숭실대학교 전자계산원 원장. 관심분야는 리엔지니어링, 소프트웨어 유지보수, 소프트웨어 재사용, 소프트웨어 재공학/역공학, 소프트웨어 테스팅