진보된 정적 분석 도구(Advanced Static-Analysis Tool)를 통한 Safety-Critical SW의 결함 검출 및 동적 테스트를 위한 대비

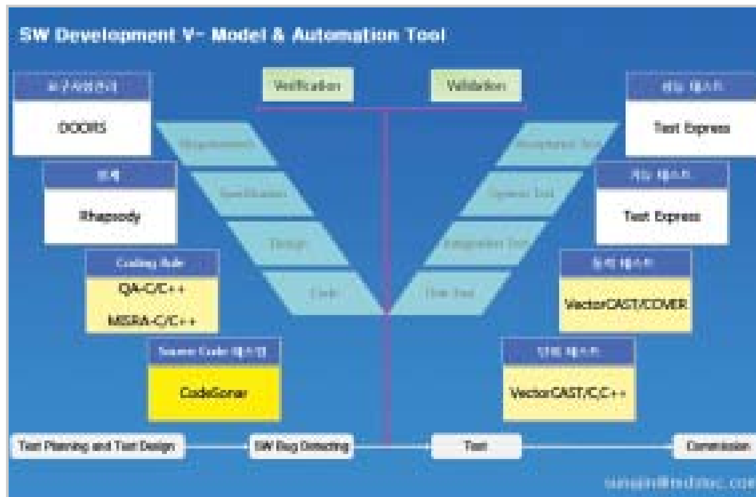
최근 도요타 사태에서 볼 수 있듯이 SW의 중요점은 굳이 언급하지 않아도 모두가 잘 알고 있는 사실이다. 그렇다면 이렇게 중요한 SW의 품질을 보증하는 방법에는 어떤 것들이 있는가? SW의 품질을 보증하는 방법에는 대표적으로 V&V(Verification & Validation)가 있다. Verification은 SW를 요구사항에 따라 구성하고 프로그램을 작성하는 과정이자 SW에서 발생할 수 있는 결함들을 찾는 과정이다. Validation은 작성된 SW가 요구사항에 따라 올바른 동작을 하고 있는지를 검사하는 것이다. 다시 말해, 개발 프로세스의 V모델에서 Verification은 V모델의 왼쪽부분을 Validation은 V모델의 오른쪽 부분을 의미한다. 이 글에서는 Validation 단계에서 발생할 수 있는 문제점들을 어떻게 Verification단계에서 대비할 수 있는지 알아보겠다.

글 : 박성진 선임컨설턴트, Test Automation사업팀 / MDS테크놀로지 www.mdstec.com

SW 안정성을 높이기 위한 노력

Safety-Critical SW에서 발생하는 결함은 단지 '상당한 골치거리'가 아닌 인명 피해를 줄 수 있는 큰 문제이다. 따라서 SW 개발자들은 이러한 결함이 시장에서 발생하는 것을 방지하기 위해 엄청난 시간을 투자하고 있다. 비록 수년간 찾아낸 잘 알려진 심각한 SW 결함들을 정의(IEEE, CWE 등)했음에도 불구하고 아직도 자동차 제어 시스템, 비행제어 시스템 및 의료기기와 같은 Safety-Critical SW에서 이러한 문제들이 계속적으로 일어난다는 것은 정말 안타까운 일이 아닐 수 없다.

그렇다면 SW 개발자들은 이러한 문제를 줄이기 위해서 어떻게 대응하고 있는가? 크게 두 가지 방법이 있는데, 첫 번째는 엄격한 테스팅 과정을 거치는 것이 있으며 두 번째 방법은 정적 분석 도구를 사용하는 것이다. 역사적으로 정적 분석 도구를 사용하는 방식은 표준 혹은 코딩 스타일 룰을 강화하기 위해 쓰여 왔다. 이러한 코딩 룰을 통한 검사는 의도가 명확하지 않은



표현식의 사용을 방지하고 또한 SW복잡도를 낮춰 현재 혹은 향후에 발생 가능할 결함을 줄여준다.

하지만 이러한 코딩 룰을 검사하는 정적 분석 방식은 SW의 문법적인 패턴만을 검사하는 intra-procedural 분석 방식을 사용하기 때문에 파일간 및 함수간의 관계에서 발생할 수 있는 결함들은 검출하기 어려우며 높은 False Positive(오검출) 비율을 보이고 있다. 이로 인해 효과적으로 문제를 찾아낸다는 것은 어려운 일일 것이다. 이러한 문제점을 극복하고자 요즘에는 진보된 정적 분석 도구(Advanced Static-Analysis Tool)가 부각되고 있다. 이러한 정적 분석 도구들은 Buffer Overrun, Null Pointer Dereferences, Memory Leak 그리고 동작 조건에 따라 결함이 발생할 수 있는 Race Conditions와 같은 심각한 결함들을 찾아 낼 수 있다. 또한 Unreachable Code, Useless Assignment, Redundant Conditions(조건문의 결과가 항상 TRUE이거나 FALSE)과 같이 개발자를 혼돈 시킬 수 있고 결함으로도 연계될 수 있는 모순된 코드를 찾아준다.

첫번째 방식인 엄격한 테스트 과정을 거치는 것의 예로는 ISO/DIS 26262, IEC 61508, DO-178B와 같은 표준들이 있다. 이러한 표준들은 SW에서 제품이 출시되기 전 해야 할 테스트를 엄격히 규정하는 것이다. 이러한 경우, 개발자들은 자신이 개발한 코드의 높은 커버리지 테스트를 해야만 하는 경우도 있다. 높은 커버리지 테스트를 하는 것은 개발자들에게 엄청난 부담을 주며 막대한 비용을 치러야 하는 문제를 안고 있다. 그러나 진보된 정적 분석 도구(Advanced Static-Analysis Tool)는 개발자들에게 높은 커버리지 테스트 시 발생할 수 있는 불필요한 부담을 제거해 주어 비용을 줄이는데 도움을 줄 수 있다. 이번 글에는 진보된 정적 분석 도구(Advance Static-Analysis Tool)들이 높은 커버리지 테스트를 보완하는 방법과 잠재적 결함을 찾는 법, 그리고 어떻게 테스트 비용을 줄일 수 있는 지에 대해 논하고자 한다.

진보된 정적 분석 도구(Advanced Static-Analysis Tool)란?

진보된 정적 분석 방식은 Model checking과 추상적인 해석과 같은 난해한 기술이 아니며 비교적 단순한 구문 문제나 코딩스타일을 찾는 것이 아닌, 심각한 코딩결함을 찾을 수 있는 새로운 기술이다. 또한 이러한 진보된 정적 분석 도구(Advanced Static-Analysis Tool)는 대부분의 빌드 시스템에 쉽게 통합이 될 수 있으며, 짧은 시간에 문제를 검출해 준다. 또한 검출된 문제는 이해하기 쉬운 리포트로 생성이 되며 False Positive(오검출) 문제까지도 관리하는 기능을 제공하고 있다.

이상적으로 가장 좋은 SW 테스트는 수많은 입력 값의 조합과 발생 가능한 모든 실행 경로를 수행 시킬 수 있는 조건들을 입력하는 것이다. 이것을 만족하는 단순한 방법은 Statement 커버리지를 확보 하는 것이다. Statement 커버리지는 모든 코드가 적어도 한번을 실행 되었다는 것을 보장 할 수 있다.

그러나 모든 개발자들이 알고 있듯, 모든 코드가 적어도 한번 실행 되었다는 것이 발생 가능성이 적은 특이한 조건에서도 문제가 발생하지 않음을 보장하지는 않는다. 그래서 Statement 커버리지를 보완하기 위해 Path 커버리지 방식이 존재한다.

이 Path 커버리지 방식은 Statement 커버리지의 단점인 발생 가능성이 적은 특이한 조건에서의 결함 검사도 가능하다. 하지만 발생 가능성이 적은 조건을 모두 테스트 하는 것은 엄청난 시간을 필요로 하게 되는 문제점을 안고 있다. 이러한 점에서 진보된 정적 분석 도구(Advanced Static Analysis Tool)가 필요한 것이다.

정적 분석은 함수 호출경로를 검사하고 개략적으로 조건문의 조건과 프로그램 상태를 고려한다. 이것으로 인해 일반적인 테스트에서 획득 될 수 있는 커버리지 보다 더 높은 커버리지상에서 발생 할 수 있는 결함 검사를 할 수 있다. 일반적인 테스트에서 높은 커버리지를 획득하기 위해서는 테스트 케이스 작성을 위한 많은 시간 투자를 해야 하지만 진보된 정적 분석 도구(Advanced Static Analysis Tool)를 사용 하면 테스트 케이스 작성에 시간을 전혀 들이지 않고도 높은 커버리지의 테스트 결과를 획득 할 수 있는 것이다. 이것이 테스팅에 필요한 시간을 줄일 수 있는 첫 번째 주요 관점이다.

또한 SW 개발 시 테스트 케이스를 작성하지 않고 단순히 개발자의 경험에 의해 결함을 찾는 국내 관행상 상당한 이점을

줄 수 있다고 생각한다. 그리고 Buffer Overrun이나 메모리 Leak과 같은 문제로 발생한 시스템 Lock-Up 및 Reset 문제의 경우 그 문제의 원인을 찾기 위해 디버깅을 해본 개발자라면 그 작업이 얼마나 많은 시간을 소비하고 있다는 것은 잘 알고 있을 것이다. 하지만 진보된 정적 분석 도구(Advanced Static Analysis Tool)는 이러한 결함들을 실제 구동되면서 발생하기 이전에 자동으로 찾아내고 그 원인을 보여줌으로써 디버깅에 필요한 시간을 줄이는 큰 이점을 가지고 있다.

어떤 방식으로 분석을 하는가?

정적 분석에는 구체적인 특정 변수 값(Specified Value) 대신에 추론 가능한 모든 변수의 값을 사용한다. 또한 이 값들에는 Symbolic Name이 주어진다. 그리고 분석에는 이 Symbol들을 이용하여 코드의 실행 경로를 추적하며 분석을 하는 것이다. 이 과정을 통해서 그 변수들이 실행 경로에서 어떠한 관계를 가지고 어떠한 영향을 미치는지 알아낼 수 있는 것이다. 예를 들어 조건에 따라 실행되는 경로가 달라지는 상황에서 특정 변수 x가 10보다 클 때 포인터 변수 p가 NULL이 될 수 있다라는 추론이 가능하며 이러한 경우 진보된 정적 분석 도구(Advanced Static Analysis Tool)에서는 Null Pointer Dereference라는 문제를 검출하는 것이다.

또한 좋은 정적 분석 도구의 경우에는 검출된 문제가 실제 문제(True Positive)인지 실제 문제가 아닌지(False Positive)를 확인하기 쉽게 문제의 원인에 대한 설명까지 보여준다. 이 부분이 개발자의 경험에 의존한 수동적인 테스팅과 도구에 의한 자동적인 정적 분석의 기초적인 차이라고 볼 수 있다. 개발자 경험에 의한 수동적인 테스팅은 단순히 테스트 케이스에서 정의된 입력 값을 통해 결과를 알아내기 때문에 정의한 테스트 케이스에 명시되지 않은 경우에 대한 문제는 확인 할 수 없는 반면 정적 분석의 경우 입력 가능한 모든 값을 통해 결과를 추론 함으로서 수동적인 테스팅 보다 훨씬 높은 커버리지상에서 발생 가능한 모든 문제를 찾아 낼 수 있는 것이다. 이 뿐만 아니라 도구에 의한 자동적인 정적 분석의 경우는 프로그램 언어의 근본적인 코딩 룰 위반과 잘못된 라이브러리 함수 사용으로 발생할 수 있는 문제까지 찾아 낼 수 있다.

다음은 정적 분석 도구에서 검출할 수 있는 중요한 결함들이다. 첫번째로 심각한 결함들을 꼽자면 시스템을 비정상적으로 종료시키거나 전혀 예상하지 못한 동작을 유발 시키는 결함들이다.

이러한 결함은 대표적으로 Buffer Overrun / Underrun, Null Pointer Dereference, Division By Zero, Uninitialized Variable, 그리고 C언어의 malloc함수와 C++ 언어의 new연산자의 오동작 및 잘못된 사용으로 발생하는 메모리 할당관련 결함인 Double Free, Use After Free, Memory Leak이 있다. 이 결함들은 대부분의 경우 결함이 발생한 위치에서 바로 문제점이 나타나는 것이 아니라 그 이후 정해지지 않은 코드의 위치에서 문제점이 나타남으로 문제의 원인을 찾기 위해 디버깅을 한다는 것은 여간 까다로운 일이 아닐 수 없다. 그리고 Dead Lock과 같은 Concurrency 결함은 스레드(Thread) 관련 라이브러리의 잘못된 사용으로 발생하게 되는데 이 결함으로는 Double Lock / Unlock, Race Conditions 등이 있다. 이러한 결함들은 동작 상황에 따라 문제의 발생 유무가 결정되므로 이 또한 디버깅하기 까다로운 문제들이다.

두번째로는 모순성 혹은 불필요한 코드로 인한 결함들이다. 이 결함들은 당장의 문제점은 아니지만 향후 잘못된 코드 이해로 인한 문제를 발생 시킬 수 있는 결함들이다. 이 결함은 대표적으로 Redundant Condition, Useless Assignments, Null Test After Dereference등이 있다.

진보된 정적 분석 도구에서 검출될 수 있는 결함 예제

그림1은 진보된 정적 분석 도구(Advanced Static Analysis Tool)중 하나인 CodeSonar에서 검출한 문제 예시이다.

그림 1에서 빨간색으로 표시된 코드는 이 문제가 발생하였을 때의 실행경로를 의미한다. 또한 왼쪽 칼럼에는 이 문제가 발생하였을 때의 조건을 표시하고 있다. 노란색 배경으로 되어 있는 부분이 이 문제가 발생한 라인이며 녹색 배경으로 되어 있는 부분은 이 문제의 발생 원인이 되는 라인이다. 이 결함은 state라는 포인터 변수가 186번 라인에서 NULL값 이라는

그림 1 CodeSonar에서 검출한 Null Pointer Dereference

것을 의미하며 왼쪽 state<=4095의 의미는 가상 메모리의 0번째 페이지를 의미한다. 문제 발생 위치에서 빨간색 코드를 따라 문제의 원인을 역추적 하게 되면 이 문제의 원인은 acquire_state()함수의 162번 라인에서 NULL을 리턴 함으로서 발생한다는 것을 쉽게 확인할 수 있다. 또한 acquire_state()함수의 첫 번째 인자인 acquire_err의 값이 acquire_state()함수의 161번 라인에서 REG_NOERROR값으로 넘어와 add_src_nodes()함수의 180번째 라인의 조건이 거짓이 되어 예외 처리를 하지 못한 근본적인 원인 또한 확인을 할 수 있다. 또한, 위에서 검출된 결함을 통해 정적 분석에서의 중요한 사실을 알 수 있다.

첫번째는 inter-procedural 기법이다. inter-procedural 기법이란 함수 호출 관계에서 참조해 오거나 리턴 되어 오는 값들을 호출 그래프에 따라서 값을 추적하는 방식이다.

다시 말해 앞서 설명한 것과 같이 이 문제의 근본적인 원인은 add_src_nodes() 함수에서 acquire_state() 함수를 호출하여 참조해 오는 acquire_err 값에 의해 발생한다는 것이다. 두번째는 실행될 수 있는 경로를 파악하고 있다는 것이다. 위 문제에는 Null Pointer Dereference 결함이 발생하지 않는 다른 실행 경로가 존재한다. 그것은 add_src_nodes()함수의 180번 라인에서 조건이 참이되는 경로이다. 만약 이러한 경로 추적에 대한 고려가 없다면 이러한 문제를 분석하지 못할 것이다.

False Positive에 대한 고찰(考察)

앞에서 알아본 것과 같이 정적 분석 도구를 이용하면 작성한 프로그램이 가지는 거의 모든 결함을 찾아내는 것은 상당히 쉬운 일이다. 하지만 모든 정적 분석 도구는 False Positive라는 오검출을 포함하고 있기 때문에 정적 분석 도구의 분석이 얼마나 효과적인 것인가는 False Positive와 False Negative(미 검출)의 비율을 얼만큼 균형 있게 가지고 있느냐 하는 것이다.

단순히 생각한다면 False Positive와 False Negative의 비율을 모두 떨어뜨리면 되지 않냐는 생각을 할 것이다. 하지만 False Negative문제를 최대한 줄이기 위해서는 함수 호출 경로 및 변수의 경우를 더 많이 고려해야 하는데 이것은 반대로 더 많은 경우의 고려로 인해 False Positive의 비율을 높이기 때문에 이 둘의 관계는 불가분(不可分)의 관계라 할 수 있다. 그러므로 프로그램의 함수 호출 관계 및 함수의 복잡도가 높아질수록 False Positive의 비율은 높아 질 수 밖에 없는 것이다.

너무 많은 False Positive문제는 개발자들에게 이 결함이 실제 문제인지 아닌지를 판단해야 하는 시간적인 손실을 가져다 준다.

또한 이것은 검출된 문제를 검토하는 과정에서 개발자들에게 심리적인 문제점도 안겨다 준다. False Positive비율이 높아 질수록 개발자들은 이것을 검토하는 것을 소홀히 함으로서 실제 문제를 False Positive로 취급할 가능성이 높아진다는 것이다. 그러므로 효율적인 정적 분석 도구의 사용을 위해서는 검출된 결함의 위험성에 따라서 그 결함을 검토하는 노력을 다르게 가져가야 하는 것이다.

예를 들어 의료기기 프로그램에서 발생한 Buffer Overrun은 사람의 생명을 위협 할 수도 있으나 게임 조정기에서 발생한 Memory Leak은 단순히 시스템을 Reset하면 되는 것과 같이 똑같은 치명적인 결함이라 할지라도 문제의 심각성은 다르다고 볼 수 있다.

이렇듯이 개발자들은 정적 분석 도구에서 검출한 문제를 확인하기 전에 검출된 문제들을 심각성에 따라 나누고 그것을 분석하는 시간을 차등적으로 분배해야 하는 과정이 필요하다. 일반적으로 심각성이 높은 결함에 대해서는 결함 검토에 소요되는 시간을 다른 결함들 보다 75~90%정도 더 많이 들여야 할 것이다.

정적 분석과 체계적인 테스팅

Safety-Critical SW 개발자들은 커버리지 테스트를 입증해야 한다. 그 커버리지는 DO-178B와 같은 인증에서는 크게 3가지 종류로 나누고 있으며 시스템의 특성에 따라 다르게 적용이 된다.

항공 표준인 DO-178B의 경우 항법장치와 같이 최고의 안정성을 필요로 하는 시스템의 경우에는 MC/DC 커버리지 100%가 요구되고 있으나 비행장치와 관계 없는 엔터테인먼트 장치에 대해서는 어떠한 커버리지도 요구 받지 않는다. 일반적인 인증규격의 가장 기초적인 커버리지인 Statement 커버리지를 획득하기 위해 시도해 본 사람이라면 100%의 Statement 커버리지를 획득하기도 쉽지 않다는 것을 잘 알 것이다. Decision 커버리지는 Statement 커버리지 보다 엄격한 커버리지로서 조건 분기에 대한 모든 경로를 테스트 해야 한다.

이것을 위해서는 반드시 모든 조건 분기의 TRUE, FALSE에 대한 테스트 케이스가 작성 되어야 한다. 그럼 이에 대한 예를 살펴보자. 아래 간단한 조건 문에 대해서 몇 개의 테스트 케이스가 필요한가?

```
if( a || b || c )
  foo();
```

표 1은 위 코드의 각 커버리지 확보를 위해 필요한 입력 값에 대한 테스트 케이스이다. 표에서 보는 바와 같이 MC/DC 커버리지까지 만족을 하기 위해서 테스트 케이스를 작성하는 일은 비교적 쉬운 일은 아니다.

그런데 만약, 프로그램 상에서 Unreachable Code가 존재

커버리지	a	b	c
Statement (1 case)	1	–	–
Decision (2 cases)	1	–	–
	0	0	0
MC/DC (8 cases)	0	0	0
	0	0	1
	0	1	0
	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

표1. 각 커버리지를 위한 테스트 케이스

하고 있다면 Decision이나 MC/DC 커버리지를 고려하지 않더라도 Statement 커버리지 조차 만족할 수 없다는 것이며, 프로그램 상에서 Redundant Condition(조건문의 결과가 항상 TRUE이거나 FALSE)결함이 존재한다면 Statement 커버리지는 만족 시킬 수는 있지만 Decision이나 MC/DC 커버리지는 만족 시킬 수 없을 것이다.

이와 같이 개발자들이 높은 커버리지를 위해 아무리 완벽한 테스트 케이스를 작성한다 하더라도 커버리지 테스트 과정에서 Unreachable Code나 Redundant Condition과 같은 문제가 발견된다면 테스트 케이스 및 코드 수정과 같은 여타 일련의 과정들을 다시 수행해야 하는 시간적 손실을 가져온다. 하지만 이러한 문제점들을 사전에 인지하고 있다면 이러한 낭비는 없게 될 것이다.

그림 2는 CodeSonar에서 검출한 Redundant Condition 결함이다. 그림 2에서 볼 수 있듯이 186번째 라인과 188번째 라인의 조건 문으로 인해 191번째 라인에서의 rest변수의 값은 적어도 3이상 일 것이다. 그러므로 191번째 라인의 조건은

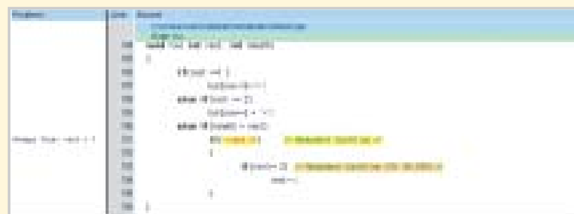


그림 2 CodeSonar에서 검출한 Redundant Condition

항상 TRUE일 수 밖에 없는 것이다.

또한 이러한 이유로 인해 193번째 라인에서도 Redundant Condition문제가 발생한다. 사실 그림 2의 예제는 비교적 쉽게 눈으로도 확인 될 수 있는 문제이다. 하지만 각 조건 문에 대한 실행 라인이 길거나 특정 함수의 Return값으로 조건을 받아 온다면 이러한 문제를 쉽게 찾아 내는 것은 상당히 힘들 것이다. 그러나 정적 분석 도구를 사용한다면 이러한 문제를 쉽게 찾아 낼 수가 있는 것이다.

결론

충분하지 못한 테스트를 거쳐져 만들어진 SW는 결함을 내제하고 있을 수 밖에 없다. 이러한 SW가 시장에 출시되고 시장에서 문제점이 발생한다면 제일 먼저 기업의 이미지에 상당한 타격을 받을 것이다.

또한 이것은 개인적인 측면에서도 개인의 경력 및 개발 역량에 대해서도 큰 오점을 남길 수 밖에 없을 것이다. 하지만 현실적으로 하루하루 개발 일정만을 맞추기에도 빡빡한 일정 속에서 충분한 테스팅을 한다는 것은 현실적으로 불가능한 일이다.

만약 CodeSonar와 같이 진보된 정적 분석 도구(Advan ced Static-Analysis Tool)를 통하지 않고 바로 Validation단계의 동적 테스트를 수행한다고 생각해보자. 많은 시간을 투여해 테스트 케이스를 작성하고 그 테스트 케이스를 통해 동적 테스트를 하던 중 Memory Leak이나 Buffer Overrun과 같은 문제가 발생을 한다면 이 문제의 발생 원인을 모른 상태에서 발생한 문제를 역추적해 가며 디버깅을 해야 하는데 이것이 개발 과정에서 대부분의 시간을 차지한다는 것은 개발자 모두가 잘 알고 있을 것이다.

또한 앞서 말한 것과 같이 Redundant Condition 결함이 동적 테스트 중에 찾아 진다면 이것에 대한 코드 수정 후 또 다시 테스트 케이스를 작성해야 하는 엄청난 시간적인 손실을 초래할 수 밖에 없는 것이다.

항공기, 자동차, 의료기기와 같이 고도의 안정성이 요구되는 분야에서는 그것의 안정성을 입증하기 위해서 각각 DO-178B, ISO/DIS 26262, IEC 61508, FDA인증이 존재하며 네가지 인증 모두 공통적으로 높은 커버리지 테스트 결과를 요구하고 있다. 이러한 요구를 만족 시키기 위해 무턱대고 도전을 한다면 엄청난 시간 투자는 물론 결국 포기를 하는 상황에까지 이를 수 있을 것이다.

손자병법에서 지피지기, 백전불태(知彼知己, 百戰不殆)라 하지 않았는가? 안전한 SW의 작성시 앞에서 언급한 예상되는 문제점을 잘 파악하고 CodeSonar와 같이 진보된 정적 분석 도구(Advanced Static-Analysis Tool)를 통해 그에 대한 대비를 한다면 안전한 SW 개발에서 불필요 하게 소요되는 엄청난 시간적 손실을 줄이는 것은 물론 요즘 이슈가 되고 있는 SW의 결함 발생 문제를 피해 갈수 있는 지름길 일 것이다. E

언제, 어디서나
빠르고 손쉽게
Embedded 를 만나자!
임베디드월드
Embedded 의 특징은 제작비용, 및 유통비용, 가격이 상대적으로 인쇄책자보다 저렴하며 수정 및 재가공이 가능합니다. 또한 멀티미디어 기능이 있어 표현이 다양하고 광고효과가 뛰어난 장점을 지니고 있습니다.
Embedded 를 www.EmbeddedWorld.co.kr에서 이용하시려면 Adobe Acrobat eBook Reader를 설치해야 합니다.
(eBook Service 참조)
인터넷에서 읽는 Embedded World, e-Magazine 서비스
☎ 02-2026-5700
www. Embeddednews.co.kr