Guest
Article

안전성이 매우 중요한(safety-critical)

소프트웨어 개발과 시험 방법의 최신 동향

본 고는 심각한 소프트웨어 결함이 양산 제품에 남아 있을 경우 발생할 수 있는 위험을 줄이기 위한 두 가지 기술을 집중 조명한다.

글: By Paul Anderson,
VP of Engineering at Grammatech
MDS테크놀로지(Grammatech 파트너사)

안전성이 중요한 장치에 소프트웨어를 점점 더 많이 사용하게 됨에 따라 소프트웨어로 인해 제품이 오작동 하는 비율도 증가 하고 있다. 1999년부터 2005년 까지 의료 장비 리콜에 대해 미국식약청(FDA)으로부터 입수한 데이터를 분석한 최근 결과에 의하면 1/3 경우가 소프트웨어 자체의 오류로 인해 발생한 것이었다. 앞으로 제품에서 소프트웨어의 비중이 지속적으로 증가하게 되고 소프트웨어가 점점 더 복잡해짐에 따라 소프트웨어로 인해 매우 심각한 결함이 발생할 가능성이 점점 높아져가고 있다.

C 및 C++ 언어는 내장형(embedded)으로 안전성이 매우 중요한 프로그래밍을 하는 경우의 거의 대부분을 차지하고 있다. 하지만 안전성이 중요한 소프트웨어를 개발하는데 있어 C 및 C++ 언어는 부족한 점이 많다. 이렇게 약점이 많은 언어가 실행을 할 때에 거의 아무런 오류 검사 없이 사용되면 비록 컴파일 시에 아무런 문제가 없다고 해도 실행 시 동작을 멈추는 프로그램을 작성하게 되기 십상이다. 프로그래밍 언어를 충분히 숙달하지 않은 상태에서 사용하는 것은 소프트웨어 안전성 문제를 야기한다는 데에 많은 사람들이 동의해 왔다. 코딩 표준은 프로그래머가 프로그램을 작성하는 방식을 제한하는 규칙을 정함으로써 이러한 문제를 해결하려는 것이다. 코딩 표준은 코드의 배치, 변수나 상수(또는 identifier)와 같은 이름의 선택, 그리고 구문 구조(syntactic construct)와 같은 주로 코드의 외관적인 측면을 다룬다.

가장 널리 쓰이는 것은 '명료성(clarity)' 인데 예를 들면 "goto 문을 사용하지 않는다"와 같은 것은 코드를 다른 프로그래머가 읽기 쉽게 하기 위한 것이다. 이것은 "코드를 작성하는 횟수보다 읽는 횟수가 더 많으므로 이해하기 쉬운 코드가 새로운 오류를 야기할 가능성을 줄인다" 라는 관찰에서 비롯되었다. 두 번째는 '예측 가능성(predictability)' 인데 이는 애매하게 해석될 소지가 있는 코드를 금지하는 것이다. 다른 것으로는 '단순성', '방어적인 프로그래밍', '표준 준수', 그리고 기타 좋은 습관들이다.

비록 많은 프로젝트들이 자체적인 코딩 표준을 가지고 진행되어 왔지만 코딩 표준의 집합이 등장하고 다양한 산업에 걸쳐 널리 받아들여진 것은 아주 최근의 일이다. 영국 자동차 업체를 대표하는 그룹인 MISRA(Motor Industry Software Reliability Association, 자동차 산업 소프트웨어 안전성 협회)는 1998년에 C 언어를 위한 지침(guideline)을 제정했다. MISRA C 언어 코딩 표준은 이제 안전성이 매우 중요한 소프트웨어 개발의 표준으로 널리 알려졌고 자동차 업계뿐만 아니라 의료 장비, 제어, 항공 등과 같은 여러 분야에서 사용되고 있다.

이것은 원래 C 언어를 위한 규칙이었으나 항공분야에서도 사용할 수 있도록 록히드 마틴(Lockheed Martin)에 의해 개발된 JSF++에서 유추된 규칙을 위주로 하여 C++ 언어를 위한 규칙으로 확장 되었다. JSF++는 F-35 타격전투기 공동개발을 위해 만들어진 항공용 C++ 언어 코딩 표준이다. 대부분의 규칙은 매우 단순해서 이를 지원하는 도구를 사용하면 이러한 규칙을 위반한 경우 쉽게 자동적으로 검출할 수 있다. 어떤 컴파일러들은 위반사례를 검사하는 명령행(command-line) 옵션을 가지고 있는 경우도 있어서 위반 사례가 검출되면 경고 메시지를 출력한다. 물론 표준에 부합하는지 여부를 검사하는 Programming Research Laboratory(PRL)사의 QAC++와 같은 정적 분석 도구들도 있다. 이러한 코딩 표준들은 유용하지만 조심해서 사용해야 한다. MISCRA 가이드라인에는 200개가 넘는 규칙이 있는데 대부분의 프로그래머는 쉽게 기억할 수 있는 것들이다. 하지만 융통성 없이 적용하거나 강요하면 오히려 생산성을 해칠 수도 있다. 또한 현학적으로 규칙을 해석하면 더욱 복잡하고 읽기 어려우며 유지관리가 더욱 어려운 코드가 만들어 질 수도 있다.

이러한 규칙에 대해 피하기 매우 어려운 비판은 코드 스타일 자체는 실질적으로 덜 중요한 사안이라는 지적이다. 실제 문제는 경쟁 조건(race condition), 리소스 누설(resource leak), 또는 버퍼 오버런(buffer overrun)처럼 프로그램이 오작동하게 만드는 것들이다. 따라서 중요한 것은 규칙에 부합하기 위해 노력하는 것보다 시험 항목(test case)을 작성하거나 오류에 대한 진단을 더욱 잘하기 위해 노력하는 것이다. 프로그래머가 코딩 스타일의 매우 단편적인 부분에 집착하면 고품질 코드를 작성하기 위해 필요한 좀 더 높은 수준의 근본적인 사항 들을 간과할 위험이 있다. 이것은 합리적인 주장이지만 코딩 표준을 완전히 버리자는 데에 동의하는 사람은 거의 없을 것이다. 오히려 가장 좋은 방법은 안전성이 매우 중요한 소프트웨어 개발에 적용되어야 하는 광범위한 원칙들이 있다는 사실을 인식하고 이러한 바탕 위에서 코딩 표준의 매우 구체적인 규칙들을 적용하는 것이다.

'10가지 강력한 규칙' 은 안전성이 매우 중요한 프로그램 개발자들이 준수해야 할 가장 중요한 원칙들을 정하기 위해 나사(NASA)의 우주선 개발에 사용하기 위해 JPL(Jet Propulsion Laboratory)의 제럴드 홀즈만(Gerald Holzmann)에 의해 만들어졌다. 이것은 단지 열 개의 규칙으로 되어 있어 매우 외우기 쉽다. 특정 언어에 대해 구체적인 규정을 하기 보다는 많은 다양한 언어에 적용하기 위해 충분히 일반적인 방식으로 표현되어 있다. 동시에 이것은 정적 분석 도구를 사용하여 규칙 위반을 검출하기 쉽도록 충분히 구체적이다. 마지막으로 이것은 프로그래밍 언어에서 사용하는 어휘나 구문적인 특성에 제한을 받지 않으며 개발 과정 자체에 대해서도 다룬다. 표 1에 전체 규칙이 나와 있다.

1	단순한 제어 흐름 구조만을 사용한다.
2	모든 루프에 제한적인 최고값을 지정한다.
3	초기화한 후에는 동적 메모리 할당을 사용하지 않는다.
4	함수는 60 라인을 넘지 않는다.

5	평균적으로 함수당 최소한 두 개의 assertion을 사용한다.
	assertion: 어떤 조건이 만족되어야 다음 코드가 실행되도록 조건을 명시하는 것,
	예) C 언어에서
	int divide(int a, int b)
	{
	assert(b != 0);
	return (a/b);
	}

6	데이터 오브젝트는 최소 범위 레벨(scope level)에서 선언한다.
7	Void가 아닌 함수의 리턴 값과 함수의 매개변수가 타당한지 확인한다.
8	전처리기(preprocessor)는 파일을 포함(#include)하거나 간단한 매크로에만 사용한다.
9	포인터의 사용을 제한한다. 역참조(dereferencing)는 단일 레벨로만 사용한다.
10	컴파일러의 모든 경고 옵션을 켜고 사용하며 소스 코드 분석기를 사용한다.

표 1. 10가지 강력한 규칙

가장 광범위한 규칙은 10번이다. 이 규칙은 자동화로 얻게 되는 엄청난 이점을 내포하고 있다. '소스 코드 분석기' 라는 것은 심각한 오류를 찾을 수 있는 고성능 '정적 분석 도구' 를 의미한다. 이 규칙에 의하면 비록 도구에 의해 통보 되는 어떤 경고가 거짓 양성의(false-positive, 오류처럼 보이나 실제로는 오류가 아닌) 경우라고 하더라도 코드에 대해 경고를 해야 한다. 거짓 양성(false-positive)인 경우는 도구가 코드를 분석하기 더욱 쉽도록 코드를 다시 작성할 것을 요구한다. 이것은 지나친 것처럼 보이지만 기술적으로 매우 확실하게 정당함을 증명할 수 있다. 만일 코드가 기계가 분석하기 어려울 정도로 복잡하다면 이는 또한 사람이 신뢰성 있게 분석하기에도 너무 복잡할 가능성이 매우 높다. 더욱이 도구가 거짓 양성으로 보고할 정도로 헷갈렸다면 이로 인해 진짜 버그를 보고하지 못할 수도 있다. 만일 코드의 정확한 동작에 사람의 생명이 달려 있다면 도구가 쉽게 그리고 정확하게 분석할 수 있도록 코드가 유지되어야 한다.

정적 분석 도구는 소스 코드에서 문제 있는 부분을 발견하기 위해 수 십 년간 사용되어 왔다. 하지만 린트(lint) 또는 이와 유사한 1세대 도구는 제한된 성능으로 인해 실용적이지 못했고 거짓 양성 비율이 높지 않았다. Grammatech 사의 CodeSonar와 같은 최신 세대의 정적 분석 도구는 최근에야 개발되었다. 이것들은 버퍼 오버런, 널 포인터 역참조, 경쟁 조건, 리소스 누설처럼 코드에 있는 심각한 오류를 찾을 수 있다. 이러한 결함은 언어의 기본적인 규칙을 위반하거나 API를 부정확하게 사용하여 발생한다. 이러한 도구는 일관성이 없거나 모순적인 가정을 하는 코드 부분을 발견할 수도 있다. 이러한 것들이 그 자체로 버그가 아닐 수 있지만 프로그래머가 코드의 중요한 특성을 이해하지 못했다는 것을 의미하는 경우가 많으므로 진짜 버그와 연관성이 있는 경우가 많다. 이러한 새롭고 향상된 정적 분석 도구들은 결함을 찾는데 매우 효과적이다. 이 도구들은 개발 과정과 쉽게 통합될 수 있고 중요하며 실행 가능한 결과를 산출한다. 이러한 도구들의 효율에 영향을 주는 핵심적인 요인은 낮은 거짓 양성 비율이다.

```
202     year = ORIGINYEAR; /* = 1980 */
203
204     while (days > 365)  /* 무한 루프에 빠질 수 있는 코드 */
205     {
206         if (IsLeapYear(year))
207         {
208             if (days > 366) /* 윤년인 경우 days가 366이 된 후 무한 루프 */
209             {
210                 days -= 366;
211                 year += 1;
212             }
213         }
214         else
215         {
216             days -= 365;
217             year += 1;
218         }
219     }
```

그림 1. 마이크로소프트사의 준(Zune) 휴대음악재생기를 멈추게 하는 버그를 포함한 코드

(그림 1)은 이러한 도구들에 의해 검출되는 결함의 예로 CodeSonar에서 실행한 화면의 일부분이다. 이 코드는 마이크로소프트의 휴대형 음악 재생기인 준(Zune)의 clock driver의 일부분에 해당하는데 이로 인해 2008년 새해 전날, 수 천 개의 준 단말기가 엄청나게 당황스런 락업(lock-up)에 걸렸는데 달력을 갱신하는 기능을 담당하는 이 부분에서 무한 루프에 빠지는 바람에 발생한 문제였다. 'days' 에서 평년인 경우 365를, 윤년인 경우 366을 빼려는 것이 프로그래머의 의도였지만

윤년의 마지막 날에 달력을 갱신하는 경우 208번째 줄의 'if (days 〉366)' 로 인해 days는 계속 366의 값을 가진 채로 무한 루프에 빠지게 되어 종료를 하지 못하게 된 것이었다.

최신 정적 분석 도구들은 컴파일러와 비슷하며 다음과 같이 동작한다. 첫째로 코드를 분석(parse)해서 전체 프로그램 모델을 만든다. 그 후 그 모델을 추상적으로 실행하여 실제 실행을 흉내 낸다. 실제 실행에서는 실제적인 입력 값과 변수 값을 사용하는 반면에 추상 실행에서는 추상적인 조건을 사용한다. 이것들은 x 〈 0, 또는 z = 2 * y 처럼 대개 프로그램에서 변수의 상태에 대한 정보를 인코딩하는 공식이다. 이러한 추상적인 실행은 실행 경로를 탐색한다. 그 과정에서 만일 이상을 발견나면 결함이 있을 수 있다는 경고를 발생시킨다. 예를 들면, 만일 분석하다가 포인터 p를 역참조하는 표현을 만나면 분석도구는 p가 NULL이 될 수 있는 경로가 있다는 것을 알고 그 표현에 대해 널 포인터 예외(null pointer exception)를 보고한다. 프로그램을 추상화하여 분석하기 때문에 기존의 시험방식으로 가능한 것보다 훨씬 더 많이 프로그램의 상태를 검사할 수 있다. 결과적으로 다른 방법으로는 찾기 어려운 문제를 발견할 수 있다. 이러한 도구의 가장 중요한 특징으로는 다음과 같은 것들로 1)회수(recall): 실제로 중요한 결함을 얼마나 잘 찾을 수 있는가의 정도, 2)정확도: 거짓 양성 결과가 나올 가능성의 정도, 3)성능: 도구를 사용하는데 필요한 시간과 공간(메모리, 하드 디스크 등의 용량) 등의 컴퓨터 자원 소요량 등이 있다.

일반적으로 이러한 요소들은 상호 배타적이다. 완벽한 회수(recall) 즉 모든 결함을 다 찾는 것이 불가능하지는 않지만 사소한 프로그램을 제외하고는 현재 기술로는 실용적이지 못하다. 왜냐하면 이렇게 하려면 엄청나게 많은 거짓 양성을 찾게 되고 성능이 많이 떨어지게 되기 때문이다. 비슷하게 어떤 도구가 정확도가 높아서 거짓 양성의 비율이 낮다면 이 도구는 정확도가 낮은 도구가 발견하는 결함을 찾지 못할 확률이 높아진다.

최신의 도구는 이러한 요소들은 모두 감안하여 사용자에게 가장 적절하도록 중도적인 입장을 취한다. 심각한 결함이 양산 단말기에 그대로 남아 있게 되었을 때 발생할 엄청난 비용과 치명적인 결과로 인해 안전성이 매우 중요한 소프트웨어의 개발자들에게 가장 좋은 도구는 설사 거짓 양성 비율이 좀 더 높고 분석 시간이 다소 길 더라도 더 많은 실제 결함을 찾을 수 있는 것들이다.

이러한 고성능 정적 분석 도구들은 안전성이 매우 중요한 소프트웨어를 개발하는 경우에 널리 사용되기 시작하고 있다. 정책 당국은 이제 이러한 도구들을 조사에 사용하기도 한다. Jetley 등은 FDA에서의 사례를 연구한 결과를 발표하였는데 단말기가 출시된 후에 오동작한 경우를 조사하기 위해 정적 분석 도구를 사용한 경우에 대한 것이었다. 이 연구에 따르면 약 20만 줄의 C/C++ 코드 조사한 결과 해당 조사와 관련된 것으로 판단되는 결함이 127개가 검출되었다. 이 중 45개는 제조사의 자체적인 내부 검사와 확인 과정에서 미리 검출되지 못한 것들이었다.

앞으로 안전성이 매우 중요한 소프트웨어 개발에는 모델 기반의 개발 방법이 사용될 것으로 보인다. 이것은 바람직한 추세로, 모델로부터 생성된 코드가 직접 작성한 코드보다 저수준 코딩 오류를 발생시킬 가능성이 작기 때문이다. 더욱 신뢰성 있는 프로그래밍 기술도 등장하고 있는데 구조적으로 정확성을 보장하는 방법이다. 하지만 산업계 전반에 널리 쓰이기 위해서는 많은 노력을 필요로 한다. 가까운 장래에 이러한 기술들로 인해 C나 C++와 같은 저수준 언어가 필요하지 않게 될 가능성은 없을 것이다.

결론적으로 코딩 표준과 함께 고성능 정적 분석 도구를 사용하는 경향은 앞으로 점점 늘어갈 것으로 확신한다.