SW개발 과정에서 효과적인
정적 분석 기법 적용방법

소프트웨어 개발자라면 누구나 소프트웨어 디펙트(Software Defect, 주1)로 고생한 기억이 있을 것이다. 예를 들어, 내가 며칠 동안 못 찾은 디펙트를 동료 개발자가 몇 분 만에 찾는 경우도 있고 복잡한 디버깅 과정을 거쳐 디펙트의 원인을 분석해 본 결과 지역변수를 초기화 하지 않아 생긴 단순한 문제도 있다.

글 : 이창호 주임 / MDS테크놀로지 TA(Test Automation)사업팀
www.mdstec.com

GUEST ARTICLE

소프트웨어는 형체가 없기 때문에 사람이 코드의 의미를 해석해 문제점이나 디펙트를 찾는다는 것은 정말 어려운 일이다. 출시 이전에 디펙트가 발견되면 금전적인 손실은 어느 정도 막을 수 있지만, 불과 몇 년 사이에 소스 코드의 크기와 복잡도가 너무나도 많이 증가했으므로 코드 상의 모든 부분을 개발자가 하나하나 테스트하고 디펙트를 찾아내 수정하는 작업은 거의 불가능하다고 볼 수 있다.

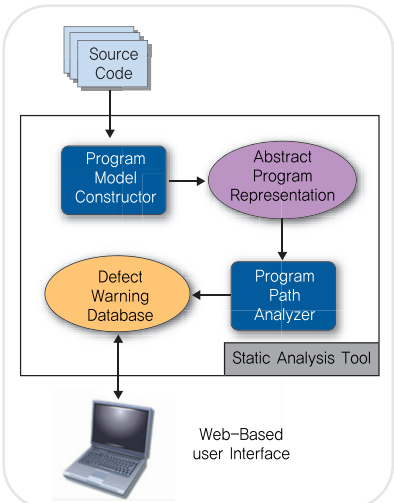

그림 1. Codesonar 분석 과정

정적 분석 기법이란

시간과 인력이 많이 소모되는 소프트웨어 테스팅 과정을 신속하고 정확하게 진행하기 위한 기법과 툴들이 연구되고 있다. 이번 기고에서 필자는 정적 분석 툴 CodeSonar를 기준으로 실행되는 정적 분석 기법(Static Analysis)에 대해 소개하도록 하겠다. 정적 분석 기법은 소스 코드 실행 없이 코드의 의미를 분석해 디

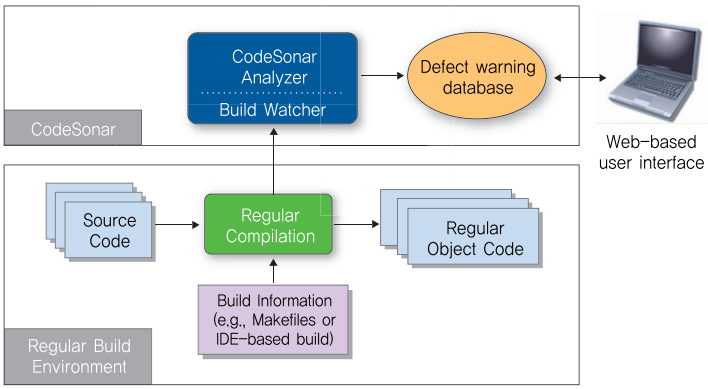

그림 2. Codesonar와 빌드 환경의 연동

```
496 _       ::NormalSum(m_ArrayFaceNeighbor,&sum);

            e:\cs_test\opensrc\opensrctest\opengl\src\wrl_viewer\src\lib3d
            Enter CVertex3d::GetMeanCurveAround() / NormalSum(CArra
245     int NormalSum(CArray3d<CFace3d> &array,
246                                         double *pSum)
247     {
248         int size = array.GetSize();
249
250         // Check
251         if(size == 0)
252             return 0;
253
254         // Only one face
255         if(size == 1)
256             return 0;
257
258         double sum = 0;
259         for(int i=0;i<size-1;i++)
260         {
261             CFace3d *pFace = array[i];
262             for(int j=i+1;j<size;j++)
263             {
264                 CFace3d *pFaceTest = array[j];
265                 if(pFace->HasNeighbor(pFaceTest))
266                     sum += SinAngle(pFace,pFaceTest);
267             }
268         }
269
270         *pSum = sum;
271         //TRACE("Value : %g\n",sum);
272         return 1;
273     }
            Exit CVertex3d::GetMeanCurveAround() / NormalSum(CArray
497         return sum/size;    /* Uninitialized Variable */
```

(248라인에서 사이즈 값이 1이 되면 256라인의 리턴문이 실행되어 270라인 pSum 변수 초기화 부분이 실행되지 않는다. 따라서 497 라인에서 sum 변수는 초기화 되지 않고 사용 되었다는 디펙트가 검출된다.)

그림 3. Codesonar 결과 화면

펙트를 찾는 방법이다. 그림 1에서와 같이 입력된 소스 코드를 Program Model Constructor가 분석하면 실행시의 동작을 표현한 중간코드로 변환된다. 이 중간코드에는 변수의 범위나 실행 경로를 추적할 수 있는 정보들이 담겨져 있다. 변환된 코드를 Program Path Analyzer가 분석해 실제 실행되는 경로를 추적하고 그 중 디펙트로 의심되는 항목을 찾아 리스트를 작성함으로써, 사용자는 웹을 통해 작성된 디펙트 리스트를 검토하고 실제 디펙트를 찾아낸다. 그림 2는 CodeSonar와 실제 빌드환경과의 연동 과정을 나타내고 있다. CodeSonar는 빌드 과정을 감시하는 기능을 제공하기 때문에 간단한 세팅만으로 실행 가능하다. 이 툴은 결과를 html 형태로 만들어내며 Html 결과에는 각 변수와 함수의 링크가 포함되어 클릭 몇 번만으로 디펙트를 추적 및 원인 분석이 가능하다(그림 3 참조).

정적 분석 기법을 적용한 코딩 단계의 세부 과정

소프트웨어 개발 과정을 크게 5단계로 요약한다면 그림 4와 같다.

개발자 입장에서 코딩단계를 살펴보면 버전관리 툴에서 체크아웃 하고 코드를 작성한 후 컴파일러를 통해 작성된 코드의 Syntax Error를 체크한다. 에러수정이 완료되면 체크-인을 실시한다.(그림 5 참조)

이 과정에 정적 분석 툴로 디펙트를 체크하는 단계를 포함시키면 그림 6과 같은

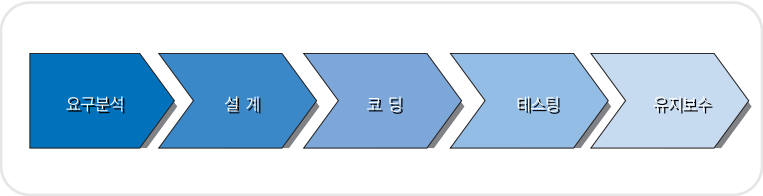

그림 4. 일반적인 소프트웨어 개발 과정

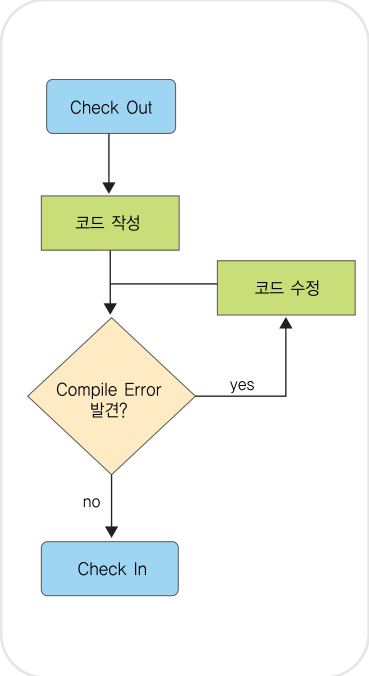

그림 5. 일반적인 코딩 과정

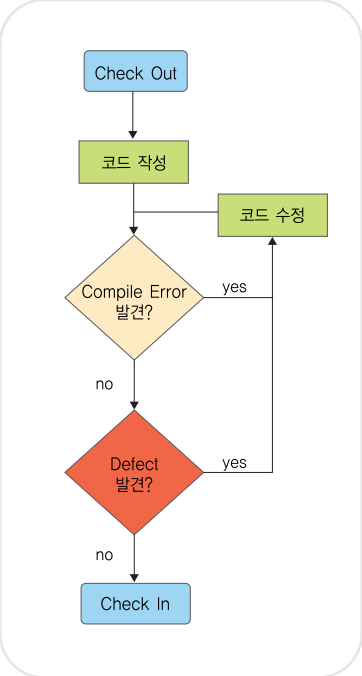

그림 6. 정적 분석 툴을 이용한 코딩 과정

진행이 이루어지며, 개발자는 syntax error가 없는 코드를 정적 분석 툴로 분석해 발견된 디펙트를 수정할 수 있다. 예를 들어, 개발자가 오늘 작성한 코드에 버퍼 오버런이 있었다고 가정할 경우, 그림 5와 같은 순서로 코딩을 하는 개발자는 해당 부분이 테스트 되지 않는 이상 버퍼 오버런의 존재여부를 판단 할 수 없다. 이 경우, 코딩 후 바로 수정하는 것보다 수정 작업이 훨씬 어려워진다. 이미 변수나 라인은 다른 곳과 연관되어 있을 확률이 크고 문제가 발견된 부분을 수정함으로써 또 다른 문제가 유발될 수 있기 때문이다. 하지만 그림 6과 같이 정적 분석 기법을 적용해 개발을 진행하는 개발자는 코딩 후 정적분석 툴을 실행함으로써 발견된 버퍼 오버런을 수정할 수 있다. 이 작업은 수정된 부분이 다른 코드에 미치는 영향을 최소화하기 때문에, 테스팅 단계의 업무 강도를 줄이고 높은 품질의 소프트웨어

Buffer Overrun	Return pointer to Freed
Buffer Underrun	Type Overrun
Division By Zero	Type Underrun
Double Free	Uninitialized Variable
Free Non-Heap Variable	Use After Free
Free Null Pointer	Double Close
Resource Leak	Double Lock
Missing Return Statement	Double Unlock
Null Pointer Dereference	Memcpy Length Unreasonable
Return Pointer To Local	Strcpy Length Unreasonable

그림 7. Codesonar에서 검출 가능한 크리티컬 디펙트 리스트

제품을 조기 출시할 수 있는 환경을 제공한다.

글을 마치며

CodeSonar가 찾아내는 디펙트 항목은 약 50여 가지이며 앞으로도 계속 추가될 예정이다. 이 가운데 발생 시 시스템에 심각한 영향을 미칠 수 있는 크리티컬 디펙트들은 그림 7과 같다.

필자는 약 3년 동안 다양한 소프트웨어 관련 기업들을 방문해 많은 라인의 소스 코드를 분석하였다. 필자의 경험에 비추어 볼 때, Buffer Overflow(주2), Buffer Underflow(주3), Resource Leak, Null Pointer Deference, Uninitialized variable의 5개 크리티컬 디펙트들이 전체의 80% 이상을 차지하고 있다. 배열의 길이를 선언할 때는 숫자를 직접 넣는 것보다는 정의된 값을 이용하고 포인터 변

수를 참조할 때 Null 유무를 체크하며, 로컬 변수를 선언할 때 초기화를 해주는 코딩 습관으로도 많은 디펙트를 줄일 수 있다. 하지만 라인이 크거나 구조가 복잡한 코드를 분석하고 검증하기 위해서는 객관적인 툴이 꼭 필요하다. 또한, 툴이 발견한 디펙트들을 분석해 발생 원인을 밝혀내고 개발자들끼리 정보를 공유함으로써 해당 디펙트들이 재발하지 않도록 관리하는 것도 정말 필요한 시스템이라고 생각한다. 무조건 만들어서 먼저 출시하고 향후에 A/S나 펌웨어 업데이트로 수정하면 된다는 생각으로 제작된 소프트웨어는 시장 경쟁에서 살아남을 수 없다. 따라서 여유 있는 개발기간을 통해 높은 품질의 소프트웨어를 조기에 출시하기 위해서는 개발에 도움을 줄 수 있는 도구들을 잘 활용해야만 할 것이다. RTime

▶ 주 1) 디펙트(Defect) : 시스템의 성능을 떨어뜨리거나 멈춤, 오동작을 유발하는 소프트웨어의 구문. 일반적으로 설계상의 오류도 포함됨.

▶ 주 2) Buffer Overflow : 배열이나 포인터의 유효 범위를 어드레스 증가방향으로 벗어나는 디펙트. CodeSonar에서는 버퍼 오버런과 타입 언더런이 해당된다.

▶ 주 3) Buffer Underflow : 배열이나 포인터의 유효 범위를 어드레스 감소방향으로 벗어나는 디펙트. CodeSonar에서는 버퍼 언더런, 타입 언더런이 해당된다.