효과적인 소프트웨어 검증을 위한 코드 자르기 도구의 개발

김동우, 박민규, 최윤자
경북대학교 IT대학 컴퓨터학부
e-mail: kdw9242@gmail.com

Code Slicing Tool for Effective Software Verification

Dongwoo Kim, Mingyu Park, Yunja Choi
School of Computer Science and Engineering, College of IT engineering
Kyungpook National University

요 약

고안전성이 요구되는 소프트웨어의 경우 극히 낮은 확률로 발생하는 오류로 인하여 전체시스템의 안전에 치명적인 상황을 야기할 수 있으므로, 철저한 안전성 검증이 요구된다. 모든 가능한 실행경로를 고려해야 하는 안전성 검증은 시간과 비용이 오래 걸리는 단점이 있다. 본 논문에서는 안전성 검증의 고비용 문제를 개선하기 위해 안전성 특질을 기준으로 코드 자르기 기법[2]을 구현한 도구를 개발하였다. 개발한 도구를 **OSEK/VDX**[1] 기반의 개방형 차량 전장용 운영체제인 Trampoline[3] 소스코드에 적용한 결과 분석 대상의 코드의 크기를 83% 줄일 수 있음을 보였다.

1. 서론

코드 자르기 기법[2]은 관심 대상의 행위와 관련된 코드만을 남기고 불필요한 코드를 제거함으로써 프로그램 본연의 행위를 유지하면서도 프로그램의 크기를 줄이는 기법이다. 고 안전성이 요구 되는 소프트웨어의 경우 극히 낮은 확률로 발생하는 오류로 인해 전체 시스템의 안전에 치명적인 상황을 일으킬 수 있으므로 철저한 안전성 검증이 필요하다. 하지만, 모든 가능한 실행 경로를 고려해야 하는 안전성 검증은 시간과 비용이 오래 걸리는 단점이 있다.

본 논문에서는 안전성 검증의 고비용 문제를 개선하기 위해 안전성 특질인 assert 함수를 기준으로 코드 자르기 기법을 적용하여 검증 비용을 절감하는 자동화 도구를 소개 한다. 개발한 도구를 사례연구에 적용한 결과 분석대상의 코드의 크기를 83% 줄일 수 있음을 보였다.

2. 연구 배경

2-1. 코드 자르기[2] 기법

다음 그림1의 예제는 총합과 평균, 팩토리얼을 구하는 코드이다. 이 코드를 반환 값을 중심으로 코드 자르기를 수행 한다고 가정하자.

코드 자르기를 위해서는 먼저 프로그램의 흐름을 분석하여야 한다. 프로그램의 흐름은 제어흐름그래프(CFG)를 이용하여 (그림 1)의 오른쪽 그래프처럼 나타낼 수 있다.

제어흐름그래프를 이용한 코드 자르기 기법을 수행하기 위해서는 먼저 기준(Criterion)이 있어야 한다. 코드 자르기를 하기 위한 기준은 C=<node, V>로 나타낸다. 이 기준에서, node는 안전성 특질이 표현된 프로그램의 문장을, V는 코드 자르기를 하기 위한 변수 집합을 지칭 한다. 다음 그림1의 코드에서는 검증하기 위한 기준=(return sum;, {sum})을 이용하여 코드 자르기를 수행한다.

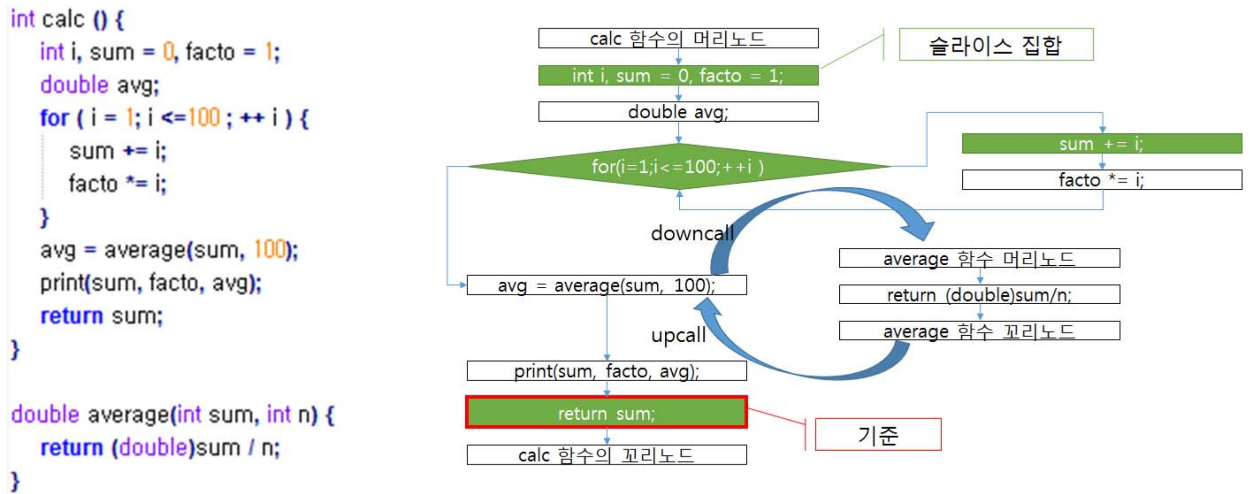
이 기준을 이용하여 기준점으로부터 기준 변수의 값을 변화시키는 문장과 변수를 추출해야 한다. 이를 위해서, 제어흐름그래프(CFG)를 기준점에서부터 위로 탐색하며 *슬라이스 집합*을 추출한다. 그 이유는 피정의 변수는 프로그램의 흐름 상 앞쪽에서 값이 정해지기 때문이다.

코드 자르기를 하는 과정은 다음과 같다.

1. 시작점 return sum;은 기준이므로 *슬라이스 집합*을 생성하고 추가한다.
2. print(sum, facto, avg) 문장은 sum의 변화와 관련이 없으므로 추출하지 않고 넘어간다.
3. avg = average(sum, 100) 문장은 avg 값의 변화에 관계가 있고, sum의 변화와 관계가 없으므로 추출하지 않고 넘어간다.
4. facto*=i 문장 또한, sum 변수의 값 변화와 관계가 없다.
5. sum+=i 문장은 sum 변수의 값을 증가하게 하므로, 추출하여야 한다. 또한 변수 i의 값이 sum 변수의 값을 변하게 하므로, i 의 값도 예의주시 하여야 한다.

이 때, 예의 주시하여야 하는 변수를 관심 변수라고 한다. 새로 추가된 관심 변수(i)의 값을 변하게 하는 문장이 있다면 그 문장도 추출하여야 한다.

6. for문에서 i의 값이 변하므로, for문 또한 추출대상에 속한다.
7. 선언문에서는 double avg; 문장은 추출된 실행문 중에서 사용되지 않아 추출하지 않아야 하지만, i와 sum은



(그림 1) 예제 프로그램과 제어 흐름도

추출된 실행문 중에서 사용하여 추출하여야 한다.

결과적으로 추출한 슬라이스 집합 = {int i, sum = 0, facto=1;, for(i=1;i<100++i);, sum+=i, return sum;}은 sum 변수와 관련된 모든 코드의 집합이다. 이 코드는 기준 변수로부터 연관이 있는 모든 코드를 추출하였으므로 컴파일 가능하다.

2-2. Downcall and Upcall

또한 (그림 1)의 코드에서 슬라이스 기준을 (print(avg); {avg})로 한다면 avg가 정의되는 avg=average(sum, 100) 문장을 슬라이스 집합에 추가 하여야 한다. 또한, avg의 값을 정의하는 average 함수의 리턴 변수에 대한 탐색이 필요하다. 그래서 먼저 average 함수의 코드 자르기를 마치고, avg=average(sum, 100) 문장을 탐색 하여야 한다. 이와 같이 호출 된 함수를 먼저 슬라이싱 하는 방법을 본 논문에서는 “Downcall Slicing” 이라고 한다.

호출 된 함수의 탐색을 마치고 나면 다시 호출 문장으로 돌아와 슬라이싱을 진행한다. 프로그램의 진행 순서상 함수 수행 이전은 함수 호출 문이 된다. 그러므로 함수의 처음까지 슬라이싱 한 후에는 자신을 호출하는 호출 문을 통하여 슬라이싱을 하여야한다. 이 슬라이싱 방법을 본 논문에서는 “Upcall Slicing” 이라고 한다.

3. 코드 자르기 도구 설계 및 구현

코드 자르기 도구는 함수 추출단계, CFG(Control Flow Graph) 생성단계, 코드 자르기 단계로 구성된다. 함수 추출 단계에서는 입력 된 프로그램의 모든 함수를 추출한다. CFG 생성 단계에서 함수 단위로 CFG를 생성한다. 마지막으로 코드 자르기 단계에서 입력 받은 기준(Criterion)으로 코드 자르기를 수행한다. 이 순서는 그림 2와 같다.

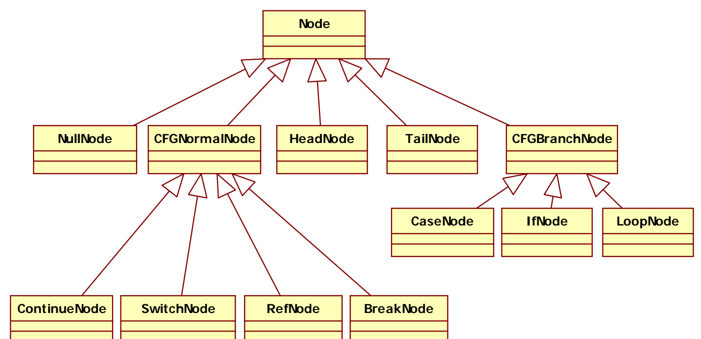
(그림 2) 코드 자르기 순서

함수 추출 시 C언어의 semantic 분석을 위해 Eclipse의 C/C++ 개발환경에서 사용되는 Eclipse CDT API를 사용하였다. Eclipse CDT API는 Expression, Name, Statement, Function 단위의 탐색을 지원한다. 다음 절 부터는 CFG의 생성과 코드 자르기를 중점적으로 설명한다.

3-1 CFG 생성

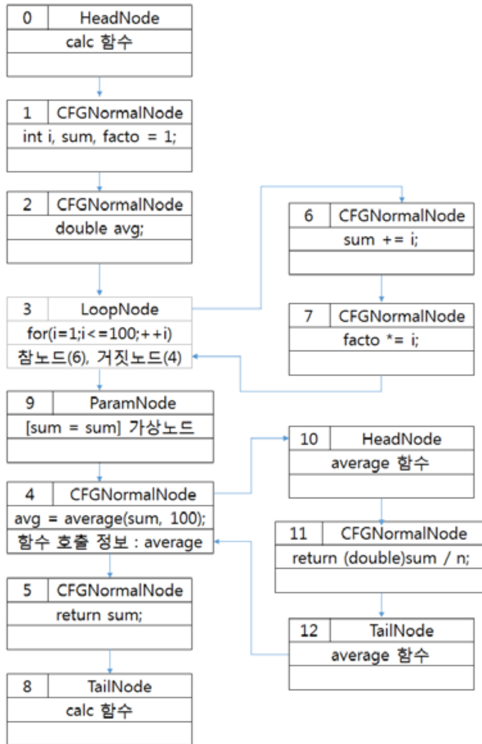
제어흐름그래프의 노드는 문장 기준으로 생성한다. If, for, while 문장으로 만든 CFGBranchNode의 다음 노드는 참 노드와 거짓 노드 2개로 구성이 되어있고, 그 밖에 다른 모든 노드는 다음 노드가 1개로 구성이 되어 있다.

ContinueNode는 가장 최근에 만난 LoopNode로 다음 노드를 연결 한다. 또한 BreakNode는 다음 노드를 가장 최근에 만난 LoopNode의 거짓 노드로 연결하거나, SwitchNode의 다음 노드와 연결한다. 또한 노드의 행위와 특성에 따라 다음 (그림3)과 같이 세분화 하였다.



(그림 3) 노드의 설계

Upcall과 Downcall 시에 함수 호출관계를 파악하기 위해 각 함수 호출 노드에 피호출 함수의 정보를 저장 한다. 각 함수 호출 노드의 다음 노드에는 피 호출 함수의 머리 노드 정보를 저장 한다. 그리고 이전 노드에는 피호출 함수의 꼬리 노드 정보를 저장 한다. 이 정보는 다음 그림 4과 같이 나타낼 수 있다.



(그림 4) 노드 구조의 모습

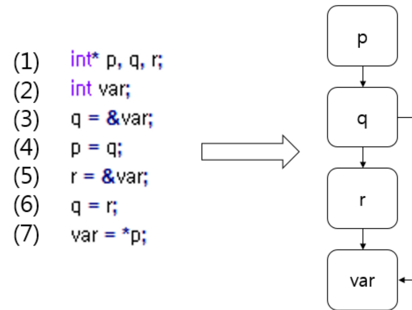
함수의 parameter는 실인자와 값이 같으므로 "매개변수 = 실인자"로 나타낼 수 있다. 그래서 함수 호출 부분 이전에 "매개변수 = 실인자" 가상 노드를 추가하여 코드 자르기를 할 때 parameter와 실인자의 관리를 쉽게 한다.

변수를 구별하기 위하여 변수의 ID와 포인터 단계를 이용한다. 우선 CDT에서 제공하는 각 변수별 고유한 ID를 이용하여 변수를 구별할 수 있다. 둘째로 포인터 단계를 사용한다. 포인터의 경우, C언어에서는 * 연산자와 &연산자의 사용에 따라, 실제로 나타내는 값이 다르다. 예를 들면 아래의 (그림 5)의 4번 문장의 p와 같이 표현 하였을 때와 7번 문장의 *p와 같이 표현한 경우는 서로 다르다. 전자는 포인터 값 p를 나타내지만, 후자는 p의 역참조된 값을 가리킨다. 이 경우 전자는 포인터 단계를 1로 두고 후자는 포인터 단계를 0으로 두어, 둘을 구별 할 수 있다. 이를 위하여 포인터 단계를 이용한다.

또한, (그림 5)와 같이 포인터가 가리키는 변수를 분석하기 위해서 포인터 관계 그래프를 만들었다. CFG 생성 중 포인터 정의 문장(포인터 정의 변수=포인터 피정의 변수)을 만나면 정의변수->피정의변수 간선을 포인터 관계 그래프에 저장한다.

포인터 관계 그래프는 hashmap(node, set<node>)을 이용하여 단방향 그래프로 구현하였다. 기존의 키 중 정의변수가 있다면 해당키의 값인 set<node>에 피정의 변수를 추가한다. 만약 정의 변수가 hashmap에 키로 존재 하지 않는다면, 새로운 set<node>를 만들어 피정의 변수를 set에 저장하고, hashmap에 정의 변수를 키로 저장한다.

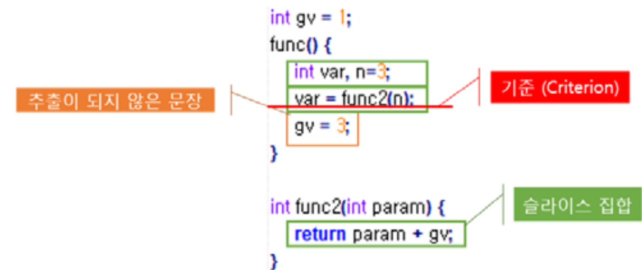
이렇게 구성된 그래프의 잎 노드에 있는 변수들은 최종 역참조 변수에 해당 한다. 코드 자르기 중 관계변수 집합에 포인터 변수가 추가되면, 포인터 관계 그래프에서 잎 노드에 있는 최종 역참조 변수도 함께 추가한다.



(그림 5) 포인터 관계 그래프 생성

3-2 코드 자르기 단계

다음 예제와 같이 기준점 아래에 있는 글로벌 변수는 코드 자르기로 완벽히 추출이 되지 않는다. 이러한 문제가 발생할 경우 코드 자르기 전에는 func 함수 호출 후 func2 함수 호출 시 gv의 값이 3이 되지만, 코드를 자른 후에는 gv의 값을 예상할 수 없게 된다.



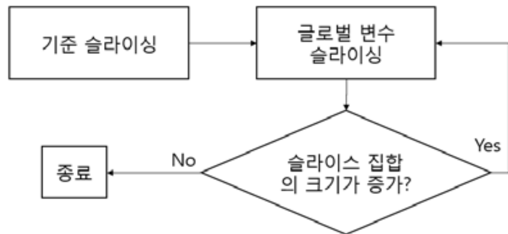
(그림 6) 코드 자르기에서 추출되지 않는 문장

이 논문에서는 그림 6과 같은 문제를 해결하기 위하여, 그림 7의 기준을 추가하여 각 CFG 마다 슬라이싱 할 필요가 있다고 연구하였다. 이 슬라이싱 방법을 글로벌 변수 슬라이싱이라 하고, 입력 받은 기준(Criterion)으로 하는 코드 자르기 방법을 기준(Criterion) 슬라이싱이라고 본 논문에서는 정의한다.

기준 = (CFG의 마지막 노드,
(관심변수 ∩ CFG에서 정의되는 글로벌 변수))

(그림 7) 글로벌 변수 슬라이싱을
하기 위한 기준(Criterion)

글로벌 변수 슬라이싱 과정을 포함한 전체 코드 자르기 과정은 (그림 8)과 같다. 먼저 기준 슬라이싱을 한 후에 (그림 6)과 같은 문제로 추출이 되지 않는 문장을 완벽히 추출하기 위해 글로벌 변수 슬라이싱을 한다. 이 과정에서 새로운 글로벌 변수가 관심 변수 집합에 추가 될 수 있다. 그래서 (그림 6)와 같은 문제로 추출이 되지 않는 새로운 문장이 생길 수 있다. 이런 문제를 해결하기 위해, 글로벌 변수 슬라이싱 과정에서 슬라이스 집합의 크기가 늘어난다면 글로벌 변수 슬라이싱 과정을 반복한다. 슬라이스 집합의 크기가 더 이상 증가하지 않는다면, 코드 추출 과정을 종료 한다.



(그림 8) 글로벌 변수 슬라이싱 과정이
포함된 코드 자르기 과정

4. 실험

차량 전장용 운영체제인 Trampoline 소스코드와 정해진 기준을 개발한 코드 자르기 도구에 입력하여, 출력된 소스코드와 기존의 소스코드의 크기를 비교해 보았다. <표 1>은 이들의 크기를 비교한 결과이다.

<표 1> 추출된 코드의 수와
기존 코드의 수(총 2437줄) 비교

기준(Criterion)	추출된 코드 (코드 수)	기존코드와 추출된 코드의 비율 (%)
tpl_h_prio	422	17.32
tpl_fifo_rw.size	438	17.98
tpl_kern.running->state	422	17.32
prio(tpl_put_preempted_proc 지역변수)	437	17.93
prio(tpl_put_new_proc 지역변수)	437	17.93
tpl_kern.running	421	17.28
tpl_ready_list.size	427	17.52
tpl_locking_depth	8	0.33

<표 1>의 결과를 비교 해 보았을 때, 코드 자르기 도구는 소스코드를 평균 83% 줄였다. 추출된 코드는 디버깅이나 정형검증처럼 시간이 오래 걸리는 작업에서 큰 효과를 발휘 할 것으로 기대된다.

더불어 수작업으로 코드 자르기를 할 때 생기는 시간문제나 정확성 문제를 자동화함으로써 해결 하였다.

5. 결론

개발한 코드 자르기 도구는 소스코드와 정해진 기준(Criterion)을 입력 받아 축약된 코드를 만들어 낸다. 이를 통하여 많은 시간이 소요되는 디버깅이나 정형검증 등의 작업에서 효율성을 증대할 수 있다.

하지만 실행문의 사이즈는 줄이는데 성공하였으나, 선언문의 크기를 줄이지는 못하였다. 선언문은 정형검증 시 많은 시간을 소모하므로, 차후 연구가 필요하다. 또한, goto 문이나 함수포인터 등의 사용이 없다는 가정에서 작동하도록 설계되었으므로 필요하다면 그에 대한 연구 또한 필요하다.

참고문헌

[1] OSEK/VDX. http://portal.osek-vdx.org.
[2] Mingyu Park, Taejoon Byun, Yunja Choi, “Property-based CodeSlicing for Efficient Verification of OSEK/VDX Operating Systems”, Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, 2012.
[3] Trampoline. http://trampoline.rts-software.org/