KIPS Tr. Software and Data Eng.
Vol.4, No.1 pp.1~8 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2015.4.1.1

CodeAnt : Code Slicing Tool for Effective Software Verification

Mingyu Park† · Dongwoo Kim†† · Yunja Choi†††

ABSTRACT

Safety critical systems require exhaustive verification of safety properties, because even a single corner-case fault can cause a critical safety failure. However, existing verification approaches are too costly in terms of time and computational resource required, making it hard to be applied in practice. In this paper, we implemented a tool for minimizing the size of the verification target w.r.t. verification properties to check, based on program slicing technique[1]. The efficacy of program slicing using our tool is demonstrated in a case study with a verification target Trampoline[3], which is an open source automotive operating system compliant with OSEK/VDX[2]. Experiments have shown enhanced performance in verification, with a 71% reduction in the size of the code.

Keywords : Program Slicing, Safety Checking, Automotive OS

CodeAnt : 소프트웨어 검증 효율 향상을 위한 코드 슬라이싱 도구

박 민 규† · 김 동 우†† · 최 윤 자†††

요 약

고안전성이 요구되는 소프트웨어의 경우 극히 낮은 확률로 발생하는 오류로 인하여 전체시스템의 안전에 치명적인 상황을 야기할 수 있으므로, 철저한 안전성 검증이 요구된다. 하지만 모든 가능한 실행경로를 고려해야 하는 안전성 검증은 고비용이 발생한다는 단점이 있다. 본 논문에서는 안전성 검증의 고비용 문제를 개선하기 위해 안전성 특질을 기준으로 코드 슬라이싱 기법[1]을 구현한 도구를 개발하였다. 개발한 도구를 OSEK/VDX[2] 기반의 개방형 차량 전장용 운영체제인 Trampoline[3] 소스코드에 적용한 결과 분석 대상의 코드의 크기를 평균 71% 줄일 수 있었고, 실제 검증을 수행했을 시에도 도구 적용 이전보다 검증 비용을 절감할 수 있었음을 보였다.

키워드 : 코드 슬라이싱, 안전성 검증, 차량 전장용 운영체제

1. 서 론

고안전성이 요구되는 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 모델 검증과 같은 안전성 검증은 고비용이 발생하는 단점이 있다. 따라서 본 논문에서는 코드 슬라이싱 기법을 검증대상에 적용해 검증대상의 크기를 줄이고자 한다.

코드 슬라이싱 기법[1]은 관심 대상의 행위와 관련된 코드만을 남기고 불필요한 코드를 제거함으로써, 검증대상 본연의 행위를 유지하면서도 대상의 크기를 줄이는 기법이다. 코드 슬라이싱 기법은 코드 내부에 나타나는 함수 간의 호출 연관관계를 중심으로 함수 간의 자료 및 제어의존성을 추적한다[1, 14]. 따라서 사용 시나리오에 따라 실행경로가 달라지는 운영체제, 내장형 시스템, 라이브러리 등의 소프트웨어에서는 전역변수의 사용으로 인해 시나리오에 종속적으로 코드 슬라이싱의 결과가 달라질 수 있다. 하지만 안전성 검증을 목표로 하는 본 논문에서는 사용 시나리오에 종속적이지 않은 포괄적인 코드 자르기의 결과가 필요하다. 또한 코드에서 나타나는 자료형의 선언 및 재정의 구문은 컴파일을 위해 반드시 필요하나, 기존의 코드 슬라이싱 방식만으로는 추출되지 않는다는 문제점이 있다[11].

본문에서 소개하는 CodeAnt에서는 시나리오에 종속적이

※ 이 논문은 2013학년도 경북대학교 전임교원 연구년 교수 연구비에 의하여 연구되었음.
※ 이 논문은 2014년도 한국정보처리학회 춘계학술발표대회에서 '효과적인 소프트웨어 검증을 위한 코드 자르기 도구의 개발'의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 경북대학교 컴퓨터학부 박사과정
†† 준 회 원 : 경북대학교 컴퓨터학부 학사과정
††† 정 회 원 : 경북대학교 컴퓨터학부 부교수
Manuscript Received : July 15, 2014
First Revision : September 11, 2014; Second Revision : November 18, 2014
Accepted : November 28, 2014
* Corresponding Author : Yunja Choi(yuchoi76@knu.ac.kr)

지 않은 포괄적인 코드 자르기 및 컴파일이 가능한 코드의 추출을 위해 전역슬라이싱 및 자료형 선언 및 재정의 구문을 추출하는 과정을 새롭게 개발하였다. CodeAnt는 ANSI C로 작성된 소프트웨어에 안전성 특질인 assert를 기준으로 코드 슬라이싱 기법을 수행해 컴파일이 가능한 코드를 출력하며, 이를 이용해 검증 비용을 절감할 수 있다. 이를 입증하기 위해 OSEK/VDX[2] 기반의 개방형 차량용 운영체제인 Trampoline[3]에 CodeAnt를 적용한 결과, 분석대상의 코드의 크기를 83% 줄일 수 있고, 실제 검증을 수행했을 시에도 도구를 적용하기 이전보다 효율적으로 검증 비용을 절감할 수 있음을 보인다.

본 논문의 나머지 부분은 다음과 같이 구성된다. 우선 제 2절에서는 사용 시나리오에 종속적인 슬라이싱 결과와 추출되지 않는 자료형 선언 구문에 대해 설명하고 이에 대한 관련연구를 소개한다. 다음으로 제 3절에서는 기존의 코드 슬라이싱 기법을 설명하고, 제 4절에서는 제 2절에서 논의한 문제들을 고려해 수정한 코드 슬라이싱 기법을 설명한다. 제 5절에서는 이를 적용한 도구 CodeAnt의 설계에 대해 설명하며, 제 6절에서는 본 도구를 적용한 사례연구에 대해 토의한다. 마지막으로 제 7절에서 본 논문의 결론을 맺는다.

2. 연구 동기 및 관련 연구

코드 슬라이싱 기법은 입력 받은 기준(criterion)을 기반으로 제어의존성(control flow dependency) 및 자료의존성(data flow dependency)을 분석해 기준과 제어의존성 및 자료의존성이 있는 구문들을 추출한다. 코드 슬라이싱을 통해 추출된 프로그램은 원본 프로그램에서 나타나는 기준과 연관된 행위를 모두 충실히 포함한다[1].

하지만 기존의 코드 슬라이싱 기법은 함수 간의 호출 연관관계만을 중심으로 슬라이싱 과정을 진행한다. 따라서 사용 시나리오에 따라 실행경로가 달라지는 소프트웨어에서는 시나리오에 따라 코드 슬라이싱의 결과가 달라질 수 있다. Fig. 1-(a)는 사용 시나리오가 아직 결정되지 않은 불완전한 소프트웨어를 나타낸다. 이 소스코드에서 만약 7번 줄을 기준으로 코드 슬라이싱을 수행한다면 사용 시나리오에 따라 종속적으로 결과가 달라질 수 있다.

```
 1 void API1(){
 2   int p = g_cnt/10;
 3   int q = 0;
 4   for(q=0; q<10; q++){
 5     print(p);
 6   }
 7   assert(p > 0);
 8 }
 9 void API2(){
10   int q = g_cnt;
11   for(q=0; q<10; q--){
12     print(q);
13   }
14   g_cnt--;
15 }
16 typedef int myInt;
17 myInt g_cnt = 0;
```

(a) Incomplete source code

```
 1 void API1(){
 2   int p = g_cnt/10;
 7   assert(p > 0);
 8 }
17 myInt g_cnt = 0;
```

(b) Sliced code result on usage scenario-*API1, API2*

```
 1 void API1(){
 2   int p= g_cnt/10;
 7   assert(p > 0);
 8 }
 9 void API2(){
14   g_cnt--;
15 }
17 myInt g_cnt = 0;
```

(c) Sliced code result on usage scenario-*API2, API1*

Fig. 1. Different slicing result caused by usage scenario

우선 사용 시나리오가 *API1*, *API2*와 같은 호출순서를 가진다면 이에 대한 코드 슬라이싱 결과는 Fig. 1-(b)와 같다. 하지만 *API2, API1*와 같은 호출순서를 가진다면 이에 대한 코드 슬라이싱 결과는 Fig. 1-(c)와 같이 나타난다. 이는 기준점을 가진 *API1*보다 *API2*가 늦게 호출될 경우, *API2*의 수행은 *API1*을 수행하는 시점에서는 영향을 주지 않기 때문이다.

이와 같은 현상은 전역변수의 사용으로 인해 발생한다. 전역변수는 인자에 의해 전달되는 변수들과는 다르게 어느 시점이든 접근/수정이 가능하다. 따라서 전역변수를 사용하는 함수들 간에는 비명시적인 자료의존성이 존재한다. 예를 들어 *API1*과 *API2*는 호출 연관관계가 존재하지 않지만, *API2*의 14번 줄에서 변경되는 전역변수 g_cnt는 *API1*의 p의 값을 변경하는 데 참여하고, p는 기준점인 7번 줄과 자료의존성을 가진다. 따라서 7번 줄과 14번 줄 사이에는 자료의존성이 존재한다.

하지만 기존 슬라이싱 연구에서 전역변수에 대해 언급한 연구[15-17]는 모두 전역변수에 대해 전역변수가 나타나는 함수의 입출력인자로만 정의하고 있다. 즉, 전역변수의 자료의존성은 인자와 마찬가지로 함수 간의 호출 연관관계만을 통해 분석된다. 그러나 사용 시나리오는 함수 간의 호출 연관관계를 변경시킬 수 있으며, 이에 따른 코드 슬라이싱의 결과 역시 달라질 수 있다.

본 논문에서는 다양한 사용 시나리오에 대해 시스템이 강건히 유지되는지를 확인하는 안전성 검증을 목표로 한다. 만약 특정한 시나리오에 대한 슬라이싱 결과 누락된 구문이 존재한다면, 이 구문으로 인해 장애(failure)에 대한 오탐, 혹은 미탐이 발생할 수 있다. 따라서 사용 시나리오에 종속적이지 않은 포괄적인 코드 슬라이싱 결과가 필요하다.

슬라이싱 연구에서 이에 대한 내용을 언급한 연구는 Larsen[15]이 있다. Larsen은 사용 시나리오가 존재하지 않는 소프트웨어에 대한 호출환경을 생성해 코드를 분석하는 방법을 제안했다. 하지만 한 클래스에 나타나는 메소드들을 목표로 분석하기 때문에 분석의 범위가 ANSI C로 작성된 코드보다 좁다. 이를 ANSI C에 적용하기 위해서는 코드에 나타나는 모든 함수를 대상으로 호출환경을 작성해야 하기 때문에 적합하지 않다.

또한 Fig. 1-(b),(c)의 코드들은 myInt 자료형이 정의되는 16번 줄이 빠져 있기 때문에 컴파일이 불가능한 코드이다. 이는 16번 줄이 추출된 구문들과는 어떠한 자료/제어의존성을 가지고 있지 않기 때문이다. 이로 인해, 실제 소스코드에 슬라이싱을 수행하는 도구 frama-C[10, 11], CodeSurfer[12, 13]는 모두 출력 결과에서 자료형 선언을 누락하고 있다. 하지만 본 논문에서는 산출되는 코드를 활용해 검증을 진행하기 때문에 반드시 컴파일 가능한 코드가 필요하다.

본 논문에서는 기존의 코드 슬라이싱 기법을 개선해 사용 시나리오에 종속적이지 않은 포괄적인 코드 자르기를 수행하고, 컴파일이 가능한 코드를 추출하는 것을 목표로 한다.

3. 정적 코드 슬라이싱

CodeAnt는 기존의 정적 코드 슬라이싱 과정을 충실히 따르며, 여기에 전역슬라이싱 과정 및 컴파일을 위해 필요한 구문을 추가하는 과정이 추가된 도구이다. 본장에서는 우선 CodeAnt의 기본이 되는 정적 코드 슬라이싱[1, 14] 및 포인터 분석 방법[18]을 설명한다.

3.1 기준기반 코드 슬라이싱

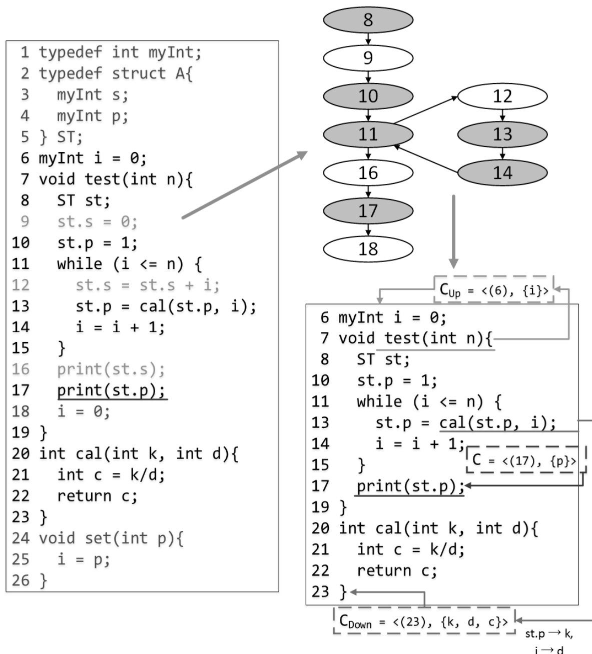

Fig. 2. Code sample and sliced code using program slicing

정적 코드 슬라이싱 기법은 프로그램 내부의 모든 구문들 사이에 나타나는 의존성 분석을 위해 우선 분석 대상 프로그램을 제어 흐름 그래프(control flow graph : CFG)로 표현한다.

• **정의 1** 함수 F에 대해 제어 흐름 그래프(CFG)는 $F_{cfg} = (N, E, s, T)$로 정의된다.
- N은 프로그램 구문 노드들의 집합이다.
- $E : N \rightarrow N$은 노드에서 노드로의 방향성 제어흐름 전이들의 집합이다.
- $s \in N$은 CFG가 시작되는 머리노드이다.
- $T \subseteq N$은 CFG의 꼬리 노드들의 집합이다. ▌

또한 노드에서 정의(*def*)되거나 사용(*ref*)되는 변수 목록을 접근하기 위한 함수를 정의하고, 해당 노드가 분기노드 아래에 있어 제어의존성이 있는 경우 해당 분기노드를 반환하는 함수를 정의한다. 해당 함수들은 아래와 같다.

• **정의 2** F_{cfg}와 F_{cfg} 내부의 노드 n이 있을 때,
- *ref*(n) : 노드 n에서 사용되는 변수들의 집합이다.
- *def*(n) : 노드 n에서 정의되는 변수들의 집합이다.
- *cd*(n) : 노드 n의 수행 가능성을 판별하는 분기노드들의 집합이다. ▌

예를 들어 Fig. 2의 노드 (12)에 대해 *ref*(12)는 {st.s, i}를 *def*(12)는 {st.s}를, *cd*(12)는 {(11)}을 각각 반환한다.

이러한 CFG F_{cfg}에 대해 정적 코드 슬라이싱은 슬라이싱 기준과 의존성을 가지고 있는 노드를 분석한다. 정적 코드 슬라이싱에 대한 슬라이싱 기준은 다음과 같이 정의된다.

• **정의 3** 슬라이싱 기준 C = <k, *V*>는 노드 k와 변수 집합 *V*의 쌍으로 구성된다. ▌

슬라이싱 기준 <k, *V*>에서 k와 *V*는 각각 슬라이싱을 수행하기 위한 기준점 노드, 관심 있는 변수를 의미한다. 정적 코드 슬라이싱은 이 노드 k를 기준으로, *ref*와 *def*를 반복적으로 수행하면서 초기 관심변수 집합을 계산한다.

• **정의 4** 초기 관심변수 집합 $R^0_C(n)$는 노드 n에 대해 아래와 같은 조건을 만족하는 모든 변수 v들의 집합이다.
1) n = k이고, $v \in V$이거나,
2) 노드 n이 노드 m의 선행 노드일 때,
   a) $v \in ref(n)$이고, $def(n) \cap R^0_C(m) \neq \emptyset$이거나,
   b) $v \notin def(n)$이고, $v \in R^0_C(m)$일 때. ▌

*정의 4*는 노드들을 방문하며 초기 관심변수 집합을 구하는 방법을 나타내고 있다. 우선 1)의 경우는 노드 n이 기준일 때를 나타낸다. 2-a)는 $R^0_C(m)$의 일부가 n에서 정의될 때를 나타낸다. 이때 정의에 참조된 변수들은 $R^0_C(n)$에 속한다. 2-b)는 $R^0_C(m)$에 속하고 n에서 정의되지 않은 변수는 $R^0_C(n)$에 속한다는 것을 나타낸다. 반대로 n에서 정의된 변수들은 $R^0_C(n)$에서는 제외된다.

Table 1의 $R^0_C(n)$ 항목은 Fig. 2의 왼쪽코드를 <(17), {st.p}>를 기준으로 *정의 4*에 따라 관심변수 집합을 계산한 결과이다. 우선 $R^0_C(17)$은 *정의 4*의 1)에 의해 {st.p}이다. $R^0_C(16)$은 st.p의 값이 정의되지 않기 때문에 2-b)에 따라 {st.p}이다. (16)의 선행노드 (11)은 후행노드로 (16)과 (12)가 있다. $R^0_C(12)$는 아직 계산 전이므로, $R^0_C(11)$을 $R^0_C(16)$만을 기반으로 계산하면 2-b)에 따라 {st.p}가 된다. $R^0_C(14)$는 2-b)에 따라 {st.p}이고, $R^0_C(13)$는 2-a)에 따라 {st.p, i}로 계산된다. $R^0_C(12)$는 2-b)에 따라 {st.p, i}가 되고, 이를 $R^0_C(11)$에 다시 반영하면, $R^0_C(11)$는 2-b)에 의해 {st.p, i}가 된다. 마찬가지로 $R^0_C(14)$ 역시 $R^0_C(11)$을 다시 반영하면 2-a), 2-b)에 따라 {st.p, i}가 되고, $R^0_C(13)$, $R^0_C(12)$, $R^0_C(11)$ 역시 같은 방식으로 다시 계산하면 {st.p, i}로 나타난다.

• **정의 5** 초기 슬라이싱 집합 S^0_C와 초기 분기 집합 B^0_C는 다음과 같이 정의된다.

- $S^0_C = \{n \in N \mid \exists m \cdot (n, m) \in E \wedge R^0_C(m) \cap def(n) \neq \varnothing\}$
- $B^0_C = \{b \in N \mid \exists n \in S^0_C \cdot b \in cd(n)\}$ ▌

초기 슬라이싱 집합 S^0_C는 모든 노드 n에 대해, 후속 노드 m의 초기 관심변수 값이 n에서 정의되는, 모든 n들을 의미한다. Fig. 2에서 *정의 5*를 통해 계산한 S^0_C는 {(10), (13), (14)}와 같다. 이를 통해 기준 C와 자료의존성을 가지고 있는 모든 노드들이 S^0_C에 포함된다. 하지만 초기 슬라이싱 집합과 제어의존성을 가지고 있는 노드들은 초기 슬라이싱 집합에 포함되지 않는다. 예를 들어 S^0_C에 포함된 노드 (13), (14)는 상위 분기노드(11)에 의해 수행이 결정되나, 이 분기노드는 S^0_C에 포함되지 않는다.

초기 분기집합 B^0_C는 S^0_C의 노드들과 제어의존성을 가지고 있는 분기노드들의 집합이다. B^0_C를 대상으로 아래와 같은 과정을 반복적으로 수행해, 앞서 분석하지 못한 분기노드 및 분기노드와 의존성이 있는 노드 역시 슬라이싱 집합에 추가한다.

• **정의 6** 분기노드와 관련된 노드 추출을 위한 반복 과정

- $R^{i+1}_C(n) = R^i_C(n) \cup \bigcup_{b \in B^i_C} R^0_{\langle b,\ ref(b) \rangle}(n)$
- $S^{i+1}_C = \{n \in N \mid (n \in B^i_C) \vee (\exists m \cdot (n, m) \in E \wedge R^{i+1}_C(m) \cap def(n) \neq \varnothing)\}$
- $B^{i+1}_C = \{b \in N \mid \exists n \in S^{i+1}_C \cdot b \in cd(n)\}$ ▌

*정의 6*에 따른 결과는 Table 1의 R^1_C, R^2_C 항목과 같다. 예를 들어, B^0_C의 (11)은 *정의 6*에 의해 S^1_C에 속한다. 다음으로 R^1_C(11-14)는 $R^0_{\langle(11),\ \{i,\ n\}\rangle}$(11-14) = {n}과 R^0_C(11-14) = {st.p, n}의 합집합인 {st.p, i, n}으로 계산된다.

본 논문에서는 *정의 4*의 초기 관심변수 집합을 구하는 과정부터 *정의 6*의 반복 과정까지를 *기준기반 슬라이싱*이라고 호칭한다. 기준기반 슬라이싱의 결과 도출되는 슬라이싱 집합은 기준 노드를 포함하지 않는다. 하지만 본 논문에서는 테스팅 과정에서 기준노드들이 필요하기 때문에 프로그램 코드를 출력할 때 기준을 추가적으로 출력하도록 한다.

기준기반 슬라이싱은 기준노드가 존재하는 함수를 대상으로 슬라이싱 집합을 구하기 때문에, 함수 간의 호출에 따라 발생할 수 있는 자료의존성을 추적하지 않는다. 예를 들어, Fig. 2에서 전역변수 i는 *test*()의 수행 이전, 전역노드(6)에서 값이 정의되기 때문에, (6)의 i는 *test*()의 i와 자료의존성을 가진다. 코드 슬라이싱 기법은 이를 위해 <(6), {i}>를 기준으로 기준기반 슬라이싱을 수행한다. 이를 *상향호출 슬라이싱*이라고 한다. 또한, (13)에서 관심변수 st.p의 값은 *cal*(st.p, i);함수의 반환 값으로 정의 된다. 따라서 st.p의 값은 *cal*(st.p, i);의 반환 값과 자료의존성을 가진다. 이러한 경우 *cal*(st.p, i);의 반환 값 역시 고려되어야 한다. 이를 위해 <(23), {k, d, c}>를 기준으로 기준기반 슬라이싱을 수행한다. 이를 *하향호출 슬라이싱*이라고 한다.

3.2 포인터 분석

포인터 변수는 참조하는 변수의 실제 주소 값을 저장하는 변수이다. 따라서 포인터가 지칭하고 있는 실제 참조 변수는 동적으로 결정될 수 있다.

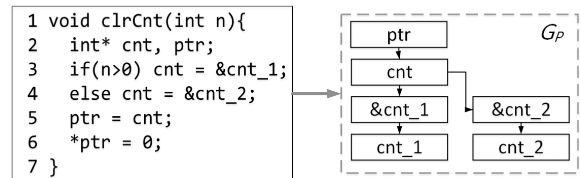

Fig. 3. Pointer relation graph

Fig. 3의 경우 포인터 변수 ptr은 인자로 전달받은 n의 값에 따라 cnt_1 혹은 cnt_2를 참조할 수 있다. 이러한 경우 포인터 변수가 지칭하는 정확한 참조 변수를 특정하는 것은 정적 분석의 관점에서는 불가능하다.

CodeAnt에서는 포인터의 분석을 위해 Anderson의 알고리즘[18]을 활용한다. 이 방법은 포인터가 지칭할 수 있는 모든 참조 변수의 목록을 작성하고, *ptr = 0;과 같은 구문이 있다면 ptr이 지칭할 가능성이 있는 cnt_1, cnt_2의 값이 모두 정의된 것으로 간주한다. CodeAnt에서도 이를 반영해 포인터를 분석한다. 예를 들어, Fig. 2의 노드 (6)에서 *def*(6)은 기존에는 {*ptr}을 반환했지만, CodeAnt에서는 {*ptr, cnt_1, cnt_2}를 반환한다.

CodeAnt에서 포인터 분석은 동적 주소 할당이 없는 코드를 가정하고 진행되었다. 이는 자동차 산업 등의 안전 중요 시스템에서 반드시 따라야 하는 MISRA-C[4] 규약에서 안전성을 위해 포인터에 동적으로 주소를 할당하는 것을 금지하기 때문이다. 따라서 포인터 변수에 동적으로 주소를 할당하거나, 포인터 변수 내부 값을 동적으로 변경시키는 사

Table 1. Iterative computation of relevant sets for C = ⟨(17), {st.p}⟩

n	8	9	10	11	12	13	14	16	17
$R^0_C(n)$	{i}	{i}	{i}	{st.p, i}	{st.p, i}	{st.p, i}	{st.p, i}	{st.p}	{st.p}
S^0_C			√			√	√		
$R^1_C(n)$	{i, n}	{i, n}	{i, n}	{st.p, i, n}	{st.p, i, n}	{st.p, i, n}	{st.p, i, n}	{st.p}	{st.p}
S^1_C			√	√		√	√		
$R^2_C(n)$	{i, n}	{i, n}	{i, n}	{st.p, i, n}	{st.p, i, n}	{st.p, i, n}	{st.p, i, n}	{st.p}	{st.p}
S^2_C			√	√		√	√		

례에 대해서는 분석하지 않았다. 만약 본 도구를 포인터에 동적 주소 할당이 발생할 수 있는 일반적인 소프트웨어에 적용했을 경우, 분석 결과가 올바르지 않을 수 있다.

4. 검증을 수행하기 위한 슬라이싱 기법의 확장

기존 코드 슬라이싱 기법의 결과는 연구 동기에서 논의된 문제점을 그대로 가지고 있다. Fig. 2에서 오른쪽 코드는 사용 시나리오가 *test()* - *set()*일 때, 기존 코드 슬라이싱을 수행한 결과에 대해 나타낸다. 이 결과에선 *set()* - *test()* - *test()*와 같은 시나리오에서 추출될 수 있는 (18, 24-26) 노드가 추출되지 않았고, 또한 자료형을 정의하는 (1-5) 노드가 추출되지 않았다. 이번 절에서는 기존 코드 슬라이싱을 수행한 뒤 이들을 추가적으로 추출하는 방법인 전역슬라이싱과 컴파일에 필요한 노드 추출 방법에 대해 설명한다.

4.1 전역슬라이싱

기존 코드 슬라이싱 기법은 앞서 논의한 바와 같이 사용 시나리오에 따라 코드 슬라이싱의 결과가 달라질 수 있다. 예를 들어 Fig. 2가 *test()* → *set()*라는 사용 시나리오에 의해 동작한다면 기존 코드 슬라이싱은 *set()* - *test()* - *test()*와 같은 시나리오에서 추출 될 수 있는 (18, 24-26)노드를 추출하지 않는다. 이 코드결과에 *set(3)* → *test(5)* 시나리오를 적용한다면, *set(3)*의 호출 시점에서 누락된 (25)구문으로 인해 i의 값이 3으로 설정되지 않는다. 이때 *test(5)*를 수행하게 되면 추출된 코드는 슬라이싱을 수행하기 전과는 다른 수행 결과를 도출한다. 즉, 슬라이싱 전/후의 행위가 일치하지 않으며, 검증 과정에서 장애의 오탐 및 미탐을 유발할 수 있다. 이는 전역변수를 통해 비명시적으로 자료의존성을 갖고 있는 노드(18, 24-26)에 대한 분석이 이루어지지 않았기 때문이다. 전역슬라이싱은 전역변수로 인해 자료연관성은 가지나 분석에서는 누락된 노드를 추출한다.

• **정의 7** 코드 슬라이싱 결과, 하나 이상의 노드가 슬라이싱 집합에 속하는 함수들의 집합 *G*가 있을 때, 전역 관심 변수 집합 GR_C는 다음과 같이 정의된다.

- $GR_C = \{\bigcup_{k \in G}(R_C(f_k) \cap \mathrm{SCOPE_{global}})\}$
- f_k는 함수 k의 머리노드이며, $\mathrm{SCOPE_{global}}$는 프로그램에서 나타나는 모든 전역 변수들의 집합이다. ▌

GR_C는 모든 코드 슬라이싱 과정이 끝난 이후 슬라이싱 집합에 포함된 함수들이 있을 때, 해당함수들의 머리노드에 등장하는 관심변수들 중에서 전역변수들을 의미한다. 예를 들어 Fig. 2에서 *정의 7*을 이용해 계산한 GR_C는 *test*()함수 머리노드의 관심변수 {i, n}에서 전역범위에 속하는 {i}이다.

• **정의 8** 전역슬라이싱 대상 집합 GR_C에 속한 각각의 전역변수 g에 대해 g의 값을 정의하는 함수 Q가 존재할 경우, 전역슬라이싱 기준 C_g는 $<n_e^Q, \{g\}>$로 정의된다.

- n_e^Q는 함수 Q의 종료 노드이다. ▌

전역슬라이싱은 C_g를 기준으로 기준기반 슬라이싱을 진행한다. 이는 GR_C에 나타나는 g에 대해, n_e^Q로부터 전역변수가 정의되는 지점을 찾고, 이를 슬라이싱 집합에 반영하기 위함이다. 이때, 함수의 머리노드에 나타나지 않는 전역변수는 이미 함수 내부에서 정의되었기 때문에 외부에서 정의된 값과는 자료의존성이 없다. 따라서 GR_C를 함수의 머리노드에 나타나는 관련변수들로 정의하였다. Fig. 2에서는 GR_C에 속한 i의 값이 *test*() 및 *set*()에서 변경되는 것을 알 수 있다. 따라서 $<(19), \{i\}>$, $<(26), \{i\}>$를 기준으로 슬라이싱을 다시 진행한다. Fig. 4는 기존 코드 슬라이싱 과정에서 전역슬라이싱 과정을 추가한 결과이다. 이를 통해 앞서 논의된 (18, 24-26)노드가 추가된 것을 확인할 수 있다.

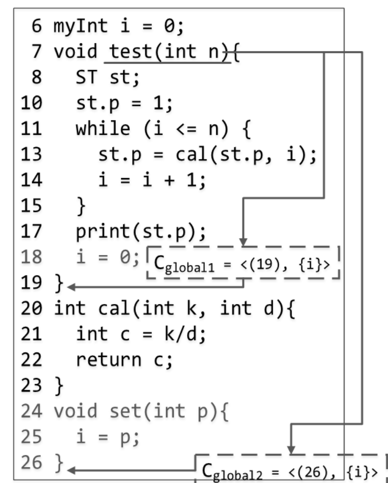

Fig. 4. Appling global slicing to sliced code

4.2 컴파일에 필요한 노드 추출

앞선 슬라이싱 단계들이 모두 끝난 이후 *Sc*에는 처음 입력받은 기준과 자료의존성, 제어의존성을 가진 모든 노드들이 포함되게 된다. 하지만 자료형을 선언하는 노드들은 *Sc*의 노드들과 자료/제어의존성이 없어 추출되지 않는다. 따라서 결과로 도출된 코드는 컴파일이 불가능할 수도 있다. Fig. 4의 소스코드는 여전히 컴파일 되지 않는 코드이다. 예를 들어 노드 (6)의 변수 i는 myInt형의 변수이기 때문에, 컴파일을 위해서는 myInt의 자료형을 정의하는 노드(1) 역시 추출되어야 한다.

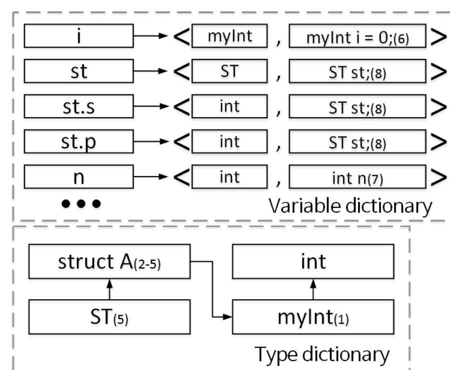

Fig. 5. Variable dictionary and Type dictionary

우선 변수의 자료 형 및 선언을 알아내고, 자료형 사이의 연관관계를 추적하기 위해 Fig. 5와 같이 변수사전과 자료형사전을 각각 생성한다. 변수사전은 변수의 자료형 및 선언되는 문장을 저장한다. Fig. 5에서 i → <myInt, (6)>의 경우, 변수 i는 myInt 자료형이며, 노드 (6)에서 선언된다는 것을 나타낸다. 자료형사전은 자료형들 간의 의존관계를 나타낸다. 예를 들어 struct A는 myInt 자료형의 s, p를 멤버 변수로 가지고 있으며, struct A를 선언하기 위해서는 myInt를 먼저 선언해야 한다는 것을 알 수 있다. 그렇기 때문에 (struct A, myInt) 전이가 자료형사전에 추가된다. 변수사전 및 자료형사전은 아래와 같은 기능을 제공한다.

- **정의 9** 변수 v와 자료형 n이 있을 때,
- *decType*(v) : 변수 v의 자료형이다.
- *decPoint*(v) : 변수 v가 선언된 노드이다.
- *tRef*(t) : 자료형 t가 참조하는 자료형의 집합이다.
- *tDef*(t)는 자료형 t가 선언된 노드이다. ▌

```
 1 void GrapNodesForCompilableCode(){
 2   set V, Ty;
 3   foreach(CFGNode n in Sc) V = V ∪ n.V;
 4   foreach(Variable v in V){
 5     Sc = Sc ∪ decPoint(v);
 6     Ty = Ty ∪ {decType(v)};
 7   }
 8   foreach(Type t in Ty){
 9     while(t != nil){
10       Sc = Sc ∪ tDef(t);
11       t = tRef(t);
12 } } }
```

컴파일에 필요한 노드 추출 단계는 다음과 같이 진행된다. 우선 Sc의 노드들에 나타나는 변수들을 찾아 변수 집합 *V*에 저장한다(3). 다음으로 변수 집합 *V*에 나타나는 각각의 변수 v에 대해 변수사전을 참조해, 변수가 선언되는 노드는 *Sc*에 포함시키고(5), v의 자료형은 자료형 집합 *Ty*에 저장

```
 1 typedef int myInt;
 2 typedef struct A{
 3   myInt s;
 4   myInt p;
 5 } ST;
 6 myInt i = 0;
 7 void test(int n){
 8   ST st;
 9   st.s = 0;
10   st.p = 1;
11   while (i <= n) {
12     st.s = st.s + i;
13     st.p = cal(st.p, i);
14     i = i + 1;
15   }
16   print(st.s);
17   print(st.p);
18   i = 0;
19 }
20 int cal(int k, int d){
21   int c = k/d;
22   return c;
23 }
24 void set(int p){
25   i = p;
26 }
```

→

```
 1 typedef int myInt;
 2 typedef struct A{
 3   myInt s;
 4   myInt p;
 5 } ST;
 6 myInt i = 0;
 7 void test(int n){
 8   ST st;
10   st.p = 1;
11   while (i <= n) {
13     st.p = cal(st.p, i);
14     i = i + 1;
15   }
17   print(st.p);
18   i = 0;
19 }
20 int cal(int k, int d){
21   int c = k/d;
22   return c;
23 }
24 void set(int p){
25   i = p;
26 }
```

Fig. 6. Extracting needed node for compiling

한다(6). 마지막으로 *Ty*에서 나타나는 각각의 자료형 t에 대해(8) 자료형 집합을 역으로 추적하며(11) 나타나는 모든 자료형들의 선언노드를 슬라이싱 집합에 추가한다(10).

Fig. 6은 Fig. 4에서 본 과정을 수행한 결과이다. 이 결과에는 컴파일에 필요한 (1-5) 역시 추출된 것을 알 수 있다.

5. CodeAnt의 설계 및 구현

CodeAnt는 ANSI C로 작성된 프로그램과 검증 특성에 기반한 슬라이싱 기준을 입력받아 코드 슬라이싱을 수행하는 도구이다. 본 도구는 CFG 생성단계, 코드 슬라이싱 단계, 코드 출력 단계로 구성된다. CodeAnt의 구조 및 절차에 대한 개요도는 Fig. 7과 같다.

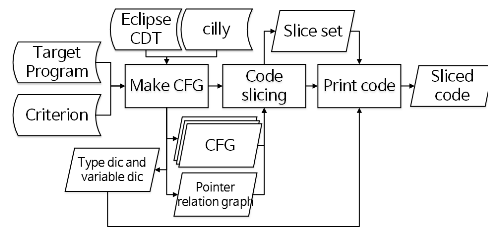

Fig. 7. Overview of tool for code slicing

분석 대상 프로그램에는 전처리가 끝난 코드가 입력된다. CodeAnt의 전처리 과정에는 CIL Driver(cilly)를 활용했다. 또한 제어 흐름 그래프를 생성할 때, 소스코드의 구조 및 의미(semantic) 분석을 위해 Eclipse의 C/C++ 개발환경에서 사용되는 Eclipse CDT API[5]를 사용하였다. Eclipse CDT API는 자동으로 AST를 생성하고, 이를 구문, 이름, 문장, 함수 단위로 탐색할 수 있도록 지원해 준다. 슬라이싱 기준에는 검증 특성이 선언된 노드 및 변수가 입력된다. 분석 대상 프로그램과 슬라이싱 기준이 입력된 이후 본 도구는 CFG 생성부터 정적 코드 슬라이싱을 시작한다.

1) CFG 생성

CodeAnt의 CFG는 *정의 1*의 형태로 작성된다. CFG를 구성하는 노드는 구현의 편의성을 위해 코드 구문뿐만이 아니라 해당 구문에 나타나는 변수, 관심변수 집합(R_C), 각 노드의 종류를 추가로 구현하였다. 또한 변수의 관리 및 구별을 위해 변수 각각에 식별자를 부여하고, 포인터 관계 그래프를 생성한다. 포인터 관계 그래프는 포인터 변수와 실참조 변수간의 지칭관계를 저장한다. 마지막으로 4.2에서 논의한 변수사전 및 자료형사전을 생성한다.

2) 코드 슬라이싱

코드 슬라이싱은 앞선 3장과 4장에서 논의된 내용들을 기반으로 구현되었다. 기준기반 슬라이싱의 경우 3장의 기존 정의들을 그대로 따르며, 이후 *정의 8*의 전역슬라이싱 과정을 추가적으로 수행한다.

3) 코드 출력

코드 출력 과정에서는 *Sc*의 각 노드에서 나타나는 변수들에 대해 변수사전과 자료형사전을 참조해, 변수들이 선언

및 변수의 자료형 선언을 S_C에 추가하고, 각 슬라이싱 과정에 사용된 기준 노드들 역시 S_C에 추가한다. 마지막으로 S_C에 나타나는 모든 노드를 코드로 출력한다.

6. 사례 연구

본 논문에서는 OSEK/VDX[2]의 표준을 기반으로 개발된 차량 전장용 운영체제 Trampoline[3]를 대상으로 사례연구를 진행하였다. 사례연구에서는 원본코드 대비 CodeAnt의 결과에 대한 효율을 비교하고, CodeAnt의 출력결과에 직접 모델검증을 수행한 결과에 대해 토의하도록 한다.

6.1 코드 슬라이싱 도구의 효율

우선 Trampoline의 전체 소스코드[1]와 검증 특성에 기반한 슬라이싱 기준을 CodeAnt에 입력해, 기존의 소스코드 대비 코드 슬라이싱 이후 코드의 크기를 비교해 보았다. 본 사례연구에서는 Trampoline의 소스코드에 남아있는 총 6개의 assert 구문을 검증 특성으로 설정하였다.

Table 2. Loc of sliced code and original code

Criterion based on verification property	Lines of executable code	Lines of total code	Efficiency of code slicing[3]	
			Executable	Total
tpl_h_prio	422 / 2437	1024 / 3294	82.68%	68.91%
tpl_fifo_rw.size	438 / 2437	1041 / 3294	82.02%	68.40%
tpl_kern.running ->state	422 / 2437	1025 / 3294	82.68%	68.88%
prio	437 / 2437	1044 / 3294	82.07%	68.31%
tpl_ready_list.size	427 / 2437	1030 / 3294	82.48%	68.73%
tpl_locking_depth	8 / 2437	571 / 3294	99.67%	82.67%

Table 2는 슬라이싱을 적용한 코드와 원본코드 그리고 코드 자르기 효율[2]을 정리한 결과이다. 여기서 실행 구문은 정의 및 선언구문을 제외한 loc를, 전체 소스코드는 이를 포함한 loc를 나타낸다. 실험 결과 CodeAnt는 전체 소스코드를 기준으로 소스코드를 평균 71% 줄일 수 있었다. 또한 검증 효율에 직접적인 영향을 미치는 실행 구문만을 대상으로 한다면 CodeAnt는 이를 평균 85.3% 줄일 수 있었다. 실행 구문의 경우와 전체 소스코드의 경우 슬라이싱 효율이 차이가 나는 것을 알 수 있는데, 이는 슬라이싱 대상 코드에서 자료형이 약 500 loc에 걸쳐 정의되어 있고, 이 자료형들이 대부분 슬라이싱 된 코드에서 그대로 쓰이기 때문이다.

6.2 슬라이싱을 적용한 코드 대상 모델 검증의 효율

다음으로 코드 슬라이싱을 통해 축소된 소스코드를 대상으로 모델 검증을 수행하고 슬라이싱을 적용한 전/후의 모델 검증 효율을 비교한다. Fig. 8은 검증과정에 대한 간략한 개요도이다.

우선 모델검증 도구는 검증하고자 하는 모델 및 모델이

1) 이는 원본코드에 cilly를 활용해 전처리과정을 수행한 코드이다.

2) *100 × (1 - Lines_of_slicedcode / Lines_of_originalcode)*

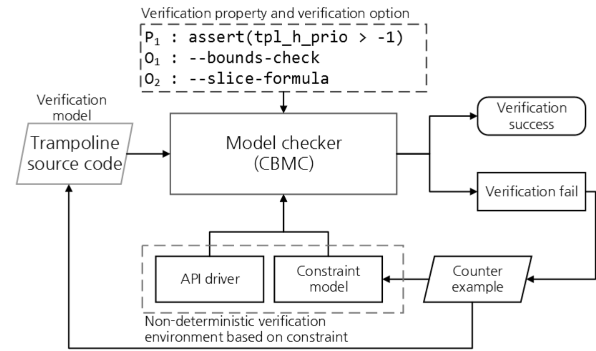

Fig. 8. Overview of verification process

만족해야 하는 검증특성 및 검증옵션을 입력받는다. 이후 모델검증 도구는 수학적 논리 증명과정을 통해 해당 모델이 검증특성을 만족하는지를 입증한다. 만약 검증이 실패한다면 이에 대한 반례를 사용자에게 제공한다. 사용자는 이 반례를 통해 소프트웨어에서 발생할 수 있는 위험성을 파악하고, 모델을 수정할 수 있다. 본 실험에서는 모델 검증기로 CBMC[6]를 사용하였다. CBMC는 ANSI C로 작성된 소프트웨어를 그대로 모델로서 활용하는 모델검증 도구이다.

CBMC를 통해 검증한 모델은 Trampoline에 코드 슬라이싱을 적용하기 전/후의 소스코드 및 이 검증모델의 환경을 담당하는 검증환경이다. 코드 슬라이싱은 Trampoline 내부의 검증 특질 중에서 assert(tpl_h_prio != 0);를 대상으로 수행되었다. 다음으로 검증환경의 경우 함수 호출기 및 제약사항 모델로 이루어져 있다. API호출기의 경우 [7]의 도구를 활용해 찾아낸, API들을 비결정적으로 호출한다. 제약사항 모델의 경우 OSEK/VDX의 명세에서 나타나는 제약사항을 모델로 작성하고, API호출기에서 호출되는 API의 순서가 제약사항을 위배하지 않도록 API 호출기에 제약을 가한다.

검증은 Fedora 19 OS, Intel Xeon 3.4GHz e3-1270 프로세서와 32GB 램으로 구성된 실험 환경에서 수행되었다. Table 3은 이에 대한 결과를 정리한 것이다.

Table 3. Verification result using model checker CBMC

Unwind \ Model	No optimization option		Lite env and using FNA[9]		Using sliced code	
	Time(s)	Mem(MB)	Time	Mem	Time	Mem
4	3,771	3,867	15	201	3	340
7	40,756	9,679	4,100	1,289	9	639
10	> 1 D	10,775	42,241	1,943	14	1,422
15	-	-	> 6 D	>7,366	17	2,176
20	-	-	-	-	56	2,741

Table 3에서 열에 표현된 모델들은 각각 원본 소스코드, 코드 슬라이싱의 결과, 기존에 발표되었던 FNA 도구[8]를 통해 최적화 된 코드[9]의 검증결과를 나타내고 있으며, 행의 경우 각각 unwind 옵션이 얼마나 적용되었는가를 나타내고 있다. unwind 옵션은 CBMC에서 논리적인 고리(loop)가 나타났을 경우 이를 몇 회 반복할 것인가에 관한 설정이다. unwind가 높아질수록 CBMC가 다뤄야 하는 모델이 복잡해지기 때문에 더 많은 비용이 소요된다.

우선 원본 소스코드의 경우 unwind 10에서 약 10GB를 넘어서는 메모리 소요와 함께 하루 이상의 검증 시간에도 검증을 마치지 못했다. 검증을 마칠 수 있었던 unwind 7의 경우에도 약 11시간의 검증 시간과, 약 9GB의 메모리가 소요되었다. 코드 슬라이싱이 적용된 코드의 경우 unwind 20에도 1분 미만, 약 2.7GB의 메모리로 검증이 가능했다. 이는 원본 소스코드를 검증한 결과뿐만이 아니라, unwind 15에서 6일 이상의 검증 시간에서도 검증을 마칠 수 없었던 기존의 최적화 도구와 비교해 봐도 효율적인 것을 알 수 있다.

7. 결 론

본 논문에서는 안전성 검증의 고비용성을 개선하기 위해 ANSI C로 작성된 코드에 코드 슬라이싱 기법을 적용하는 도구 CodeAnt를 소개했다. 본 도구는 기존 코드 슬라이싱에서 고려되지 않았던 비명시적 자료연관성 및 컴파일 가능성을 반영하기 위해 전역슬라이싱 기능과 컴파일에 필요한 노드를 추출하는 기능을 확장했다.

CodeAnt는 소스코드와 안전성 특질과 연관된 기준을 입력 받아 기준과 연관된 코드만을 추출한다. 이를 통하여 많은 시간이 소요되는 디버깅이나 정형검증 등의 작업에서 효율성을 증대할 수 있다. 본 논문에서는 실제로 이를 입증하기 위해 차량용 운영체제 Trampoline에 본 도구를 적용해, 분석대상의 코드의 크기를 약 71% 줄일 수 있었고, 추출한 코드를 기반으로 모델 검증을 수행한 결과 검증 비용이 실제로 절감되었음을 보였다.

본 코드 슬라이싱 도구는 차량용 운영체제 Trampoline뿐만이 아니라 ANSI C로 작성된 다양한 소프트웨어에 활용될 수 있는 기반도구이다. 향후에는 본 도구를 좀 더 다양한 소프트웨어에 확대해 적용하고, 이에 대한 안전성 검증을 수행하고자 한다.

References

[1] M. Weiser, Program slicing, IEEE Transactions on Software Engineering, SE-10(4), 1984.
[2] OSEK/VDX[Internet], http://portal.osek-vdx.org.
[3] Trampoline[Internet], http://trampoline.rts-software.org.
[4] MISRA-C[Internet], http://misra.org.uk.
[5] Eclipse CDT[Internet], http://www.eclipse.org/cdt.
[6] E. Clarke, D. Kroening and F. Lerda, "A tool for checking ANSI-C programs," in *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* pp.168-176, 2004.
[7] D. Kim, M. Park, and Y. Choi, "A function finder for property-based extraction of test target functions," in *The 39th Conference of the KIPS*, pp.954-957, 2013.
[8] Z. Lu and Y. Choi, "A Function Network Analyzer for Efficient Analysis of Automotive Operating System," in *The 39th Conference of the KIPS*, pp.972-975, 2013.
[9] M. Park, T. Byun, and Y. Choi, "Property-based Code Slicing for Efficient Verification of OSEK/VDX Operating Systems," in *Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems*, pp.305-319, 2012.
[10] frama-C[Internet], http://frama-c.com.
[11] B. Monate and J. Signoles, "Slicing for security of code," in *First International Conference on Trusted Computing and Trust in Information Technologies*, pp.133-142, 2008.
[12] P. Anderson and T. Tim, "Software inspection using codesurfer," in *Proceeding of the first workshop on inspection in software engineering,* 2001.
[13] The Wisconsin Program-Slicing Tool[Internet], http://research.cs.wisc.edu/wpis/slicing_tool.
[14] F. Tip, "A survey of program slicing techniques," Journal of programming languages, pp.121-189, 1995.
[15] L. Larsen, and M. J. Harrold, "Slicing object-oriented software," Software Engineering 1996, *Proceedings of the 18th International Conference on. IEEE*, 1996.
[16] P. Anderson, R. Thomas, and T, Tim, "Design and implementation of a fine-grained software inspection tool" Software Engineering, IEEE Transactions on, pp.721-733, 2003.
[17] H. Thiagarajan, et al., "Bakar Alir: Supporting Developers in Construction of Information Flow Contracts in SPARK," Source Code Analysis and Manipulation, IEEE 12th International Working Conference on 2012.
[18] L. O. Andersen, "Program analysis and specialization for the C programming language," Diss. University of Cophenhagen, 1994.

박 민 규
e-mail : pqrk8805@hanmail.net
2011년 경북대학교 컴퓨터학부(학사)
2013년 경북대학교 전자전기컴퓨터학부(석사)
2013년~현 재 경북대학교 컴퓨터학부 박사과정
관심분야 : 컴포넌트 기반 개발방법, 컴포넌트 모델 최적화

김 동 우
e-mail : kdw9242@gmail.com
2011년~현 재 경북대학교 컴퓨터학부 학사과정
관심분야 : 소프트웨어 안전성 분석, 정적 분석, 정형 검증

최 윤 자
e-mail : yuchoi76@knu.ac.kr
2003년 미국 미네소타대학 전산과(박사)
2003년~2006년 독일 프라운호퍼연구소 연구원
현 재 경북대학교 컴퓨터학부 부교수
관심분야 : 소프트웨어 안전성 분석, 정형 검증, 모델기반 개발방법론