코드 정형검증을 위한 특성기반 코드추출기

박민규*, 최윤자*, 김진삼**
*경북대학교 소프트웨어안전공학연구실
**한국전자통신연구원
e-mail : pqrk8805@gmail.com

A Property-based Code Extractor for Formal Code Verification

Min-Gyu Park*,Yunja Choi*, Jinsam Kim**
*Software Safety Engineering Lab., Kyungpook Natl. Univ.
**Electronics and Telecommunications Research Institute

요 약

안전중요 소프트웨어 코드의 검증은 1%의 잠재적 가능성을 가진 오류조차 허용하지 않는 철저한 검증방식을 요구한다. 이러한 요구에 부응하여 최근 수학적 모델을 사용한 정형검증 기법이 코드검증에 활발하게 적용되고 있으나, 코드의 복잡도와 크기의 증가에 따른 검증비용의 기하급수적 증가가 해결과제로 부각되어왔다. 본 연구에서는 검증하고자 하는 특성을 중심으로 검증대상 코드를 추출, 정형검증의 대상을 자동으로 축소하는 코드추출기를 개발하였다. 개발된 코드추출기는 자동차 전장용 운영체제의 검증에 보조적으로 활용되어 검증비용을 90% 이상 절감하고 검증 사용성을 높이는데 기여하였다.

1. 서론

자동차, 교통신호제어기, 의료기기 등에 탑재되어 시스템을 조종하는 소프트웨어는 시스템 안전에 결정적인 영향을 미치는 안전중요소프트웨어이다. 이러한 소프트웨어는 철저한 검증과정을 거쳐 1%의 잠재적 위험요소도 사전에 식별해 낼 수 있어야 하며, 기존 실행시간 테스팅과는 차별화된 검증기법을 요구한다.

정형검증기법[1]은 검증 대상소프트웨어를 수학적인 모델로 변환하여 완전검증을 실시하는, 안전중요 시스템의 검증에 적합한 기법이다. 하지만, 기존의 테스팅 기법과는 달리 모든 가능한 경우를 철저히 검증하는 기법의 특성상, 검증에 많은 자원이 소요되는 단점이 있다. 본 논문에서는 정형검증 기법을 코드검증에 적용하는 데에 있어서 소요되는 자원을 최소화할 수 있도록 코드에서 검증대상을 자동추출하는 코드 추출기법을 소개하고 개발된 코드 추출기를 사용하여 검증효율을 높인 사례를 소개하고자 한다. 코드추출기는 기존의 코드분석기 Understand 의 분석 데이터베이스를 활용하여 검증하고자 하는 특성과 연관된 코드만을 자동추출한다. 개발된 코드추출기는 자동차 전장용 운영체제의 검증에 활용되어 검증비용을 90% 이상 절감하였다.

2. 정형검증과 문제점

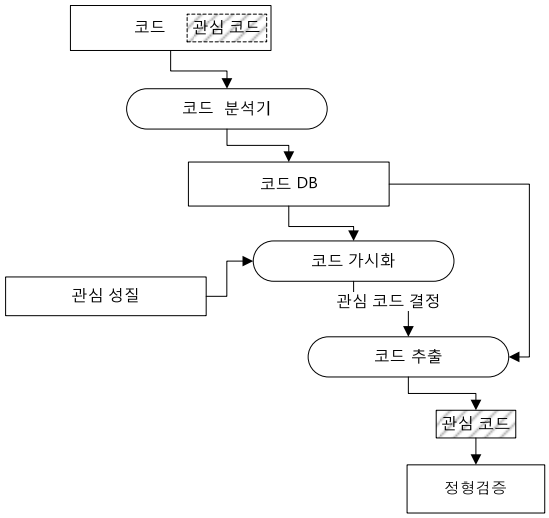
정형검증기법은 검증대상과 검증하고자 하는 속성을 논리식과 같은 수학적 분석이 가능한 표현방법으로 모델링하고, 검증대상이 검증속성을 만족하는지를 확인하는 방법이다. 정형검증은 수학적인 증명방법을 통해 검증을 수행하기 때문에 테스팅에 비해 검증의 정확도가 높다. 따라서 자동차나 항공기와 같은 고도의 신뢰성과 안전성을 요구하는 시스템 검증에 주로 적용된다. 그러나 정형검증은 검증대상을 수학적으로 모델링하는 것이 어렵고, 검증모델을 다시 구현모델로 변환해야 하며 이 과정에서 오류가 유입될 수 있는 문제가 있다.

이러한 문제를 해결하기 위해 정형검증도구 SPIN[2]은 C 언어와 유사한 Promela 라는 정형언어를 제공하고 있으며, ANSI-C 코드를 부울식으로 자동변환하여 검증하는 CBMC 와 같은 도구들이 개발되었다. 하지만 이러한 자동변환 도구들은 정형검증 기법의 사용성을 높인다는 장점이 있으나, 검증대상의 크기가 증가함에 따라 검증비용이 기하급수적으로 증가하는 상태공간폭발(state space explosion) 의 문제는 여전히 해결해야 할 과제이다. 특히 CBMC 등을 이용해 추상화 수준이 낮은 프로그램코드를 검증할 때는 이러한 문제가 더욱 심화된다.

3. 특성기반 검증

3-1. 접근방식

본 연구에서는 코드검증시의 검증비용을 줄이기 위하여 검증속성과 관련된 요소들만을 추출해 검증대상의 크기를 줄이는 방법을 제시하고, 이를 적용, 구현하였다.



(그림 1) 특성기반 코드 추출 프로세스

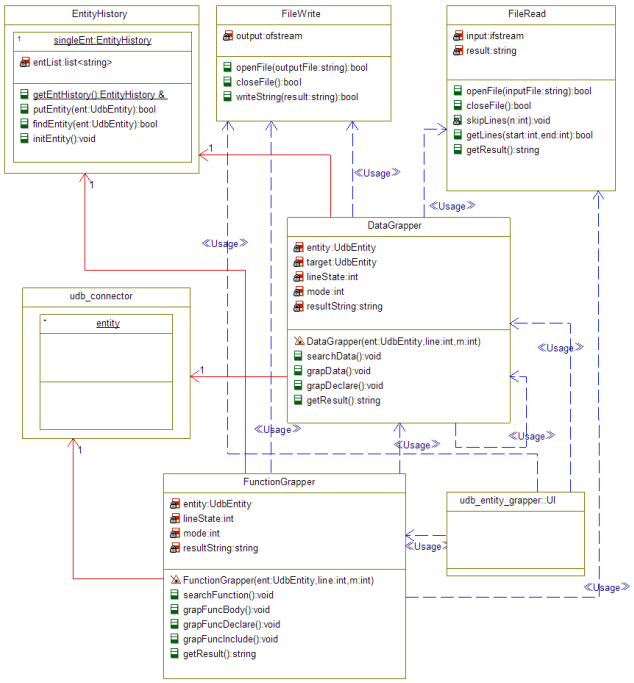
(그림 1) 은 특성기반 코드 추출을 위한 전체적인 프로세스를 도식화한 것이다. 우선 관심성질과 밀접한 관련이 있는 개체를 추출하기 위해 코드 내부 개체의 의존성 관계를 코드 분석기를 통해 분석해 데이터베이스로 구축한다. 그리고 검증하려고 하는 범위와 관련된 관심성질을 정리하고 관심 성질과 밀접하게 연관된 개체를 파악한다. 이를 위해 데이터베이스에 구축되어 있는 개체 사이 관계를 그래프로서 가시화 하고, 이를 통해 관심 성질에 밀접한 관련이 있는, 즉 추출할 개체를 결정한다. 추출해야 될 대상의 개체가 결정이 되면 데이터베이스에서 이와 연관관계를 가지고 있는 개체들을 탐색하고, 그 중에서 깊은 의존관계를 가지고 있어서 추출할 필요성이 있는 개체를 추려낸 후, 이를 추출한다.

코드개체간 의존성분석을 위해 본 연구에서는 Scitools 사의 Understand[3] 라는 코드분석기를 사용하였다. 코드추출기는 Understand 의 분석된 DB 를 활용해 검증특성과 밀접한 관련이 있는 개체를 추출한다.

3-2. 설계

Understand 는 코드 의존성 분석, 개체간의 연관성에 관한 데이터베이스 구축, 코드 가시화를 기본적인 기능으로 제공하기 때문에 코드분석 결과를 DB 로 구축하는 과정은 Understand 를 사용해 자동화 할 수 있다. 코드 추출기는 전체적으로 Understand 의 데이터베이스를 검색해 관심 성질과 일차적인 관계를 가진 개체들의 연관관계를 검색하고 밀접한 의존성이 있어 추가적으로 추출해야 될 개체가 있는지를 판단, 추출하는 전반의 행동을 수행한다. 관심 성질과 일차적인 관계를 가진 개체들은 가시화된 코드 의존성 그래프를 이용해 선별하였으며, 선별된 개체들과의 의존성에 의해 추출이 필요한 개체의 선택은 Understand 의 API 를 이용해 Understand 의 데이터베이스에 저장된 연관관계를 직접 검색해 수집했다.

다음의 (그림 2)는 코드 추출기의 전체적인 구성을 도식화하고 있다.



(그림 2) 코드 추출기의 구성

코드 추출기는 Understand 의 데이터베이스와의 연결을 담당하는 udb_connector, 파일의 I/O 를 담당하는 FileWrite 와 FileRead, 싱글톤패턴으로 정의되어 프로그램 수행 도중 DataGrapper 와 FunctionGrapper 의 행동에서 개체의 검색여부를 저장하는 EntityHistory, 개체와 의존성 개체 사이의 연관관계로 구성된 트리를 전체적으로 검색하고 추출대상의 소스를 추출하는 DataGrapper 와 FunctionGrapper 로 나누어 구성되었다.

```
//File.c
void f1();
void f2();
typedef struct{
   int data;
   char data2;
} s1;
```

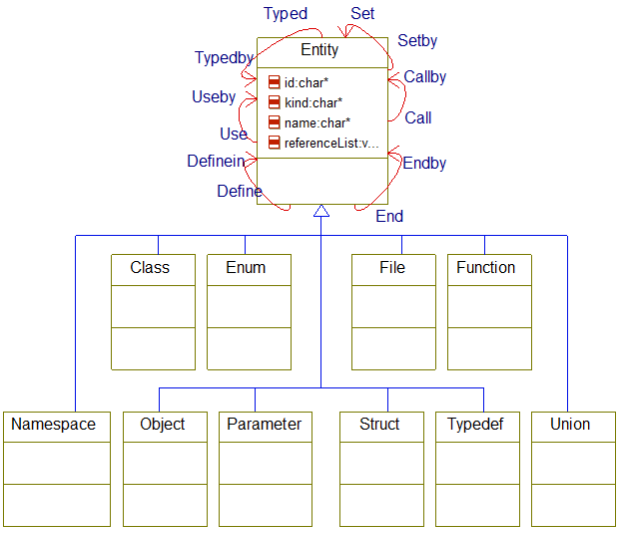
```
void f1(){
   f2();
}

void f2(){
   s1 instant;
}
```

[예제 1] 간단한 C Code

예를 들어 [예제 1]에서 f1 과 의존성이 있는 개체를 추출 할 때 개체간 행동을 보자면, 우선 UI 개체는 f1 을 대상으로 하는 FunctionGrapper 를 생성한다. FunctionGrapper(f1)는 f1 과 호출관계에 있는 f2 를 추출하기 위해 f2 를 대상으로 FunctionGrapper 를 생성하고, Function Grapper(f2)역시 f2 와 의존관계가 있는 s1 추출을 위해 s1 가 대상인 DataGrapper 를 생성한다. 그 후 DataGrapper(s1)는 FileRead 를 통해 s1 을 추출하고, FunctionGrapper(f2)와 FunctionGrapper(f1) 역시 각각 FileRead 를 통해 f2 와 f1 을 추출한다. 마지막으로 FileWrite 를 사용해 모든 추출된 내용들을 출력한다.

3-3. 구현

Understand 는 코드를 “Entity”와 “Reference”라는 개념을 기반으로 분석한다.



(그림 3) Entity 계층구조(전체 C/C++기준)

(그림 3) 과 같이 Understand 에서 Entity 는 개체 사이의 관계인 Reference list, Entity 의 고유식별자, 종류, 이름 등의 자료구조를 포함하고 있다. Entity 의 종류는 class, enum, file, function 등이 있으며 이런 성질을 이용해 Reference 를 탐색하고 개체의 종류를 구분해 개체와 연관이 있고 그 중 원하는 개체만을 접근할 수 있다.

이 절에서는 개체 사이의 연관관계를 추적, 추출하는 핵심적인 기능인 FunctionGrapper 와 DataGrapper 에 대한 구현을 기술한다. DataGrapper 와 FunctionGrapper 는 (그림 3)의 Entity 계층구조를 참조하여 구현되었다.

DataGrapper 는 대상 개체와 자료 의존성이 있는 개체의 추출을 담당한다. 기본적으로 대상 개체의 연관관계가 "Define"일 때와 "Typed"일 때 탐색을 수행한다. 대상개체와 연관된 개체를 추출하는 DataGrapper 의 searchData()에 대한 간략한 알고리즘은 다음과 같다.

```
While(!end of Target entity's reference list){
        rEntity = reference list[i]'s related entity;

        if(reference list[i]'s kind == Typed ||
        reference list[i]'s kind == Define){
            if(rEntity is not in history){
                entHistory->putEntity(rEntity);
                DataGrapper dataGrapper(rEntity);
                dataGrapper.searchData();
                dataGrapper.grapData();
                resultString+=dataGrapper.getResult();
            }
        } i++;
}
```

우선 searchData()는 수집대상개체와 자료의존성이 있는 개체인 rEntity 의 수집을 위해 rEntity 를 대상으로 새로운 dataGrapper 개체를 생성한다. 생성된 dataGrapper 는 rEntity 와 자료의존성이 있는 개체의 수집을 위해서 searchData()를 수행한 뒤 grapData()를 호출해 rEntity 를 추출한다.

FunctionGrapper 는 대상 개체가 함수 등 행동개체인 경우를 담당하며, 기본적으로 대상개체의 연관성 목록 중에서 연관성의 종류가 "Call"일 때와 대상개체와 의존관계에 있는 자료형을 추출할 때 탐색을 수행한다. FunctionGrapper 의 searchFunction()에 대한 간략한 알고리즘은 다음과 같다.

```
DataGrapper dataGrapper(Target entity);
dataGraper.searchdata();
resultString += dataGrapper.getResult();
While(!end of Target entity's reference list){
        rEntity = reference list[i]'s related entity;
        if(reference list[i]'s kind == Call){
            if(rEntity is not in history){
                entHistory->putEntity(rEntity);
                FunctionGrapper funcGrapper(rEntity);
                funcGrapper.searchFunction();
                resultString+=funcGrapper.getResult();
                funcGrapper.grapFuncBody();
            }
        } i++;
}
```

우선 searchFunction()은 수집대상개체와 자료의존성이 있는 자료형 개체의 수집을 위해 수집개체를 대상으로 dataGrapper 개체를 생성하고, dataGrapper 의 searchData()를 수행한다. 그 후 searchFunction()은 대상개체와 호출관계인 rEntity 의 추출을 위해 rEntity 를 대상으로 하는 functionGrapper 개체를 생성한다. 생성된 functionGrapper 는 rEntity 와 호출관계에 있는 다른 개체들의 추출을 위해 searchFunction()을 수행한 후, grapFunctionBody()를 통해 rEntity 를 추출한다.

(그림 4) 코드 추출기의 화면

(그림 4)는 코드 추출기의 구동 화면이다. 우선 Understand 에서 분석된 *.Udb 파일을 불러오고 추출하고자 하는 개체를 입력한다. 프로그램에는 Dependent Data 와 Entity's Body 두가지 모드가 있는데 Entity's Body 는 연관성이 있는 Entity 의 Body 를 추출하고, Dependent Data 는 Entity 와 의존성이 있는 자료형 개체를 같이 추출한다. 추출결과는 텍스트 창에 출력되며, 결과물을 복사해서 사용하거나 파일로 바로 저장할 수 있다.

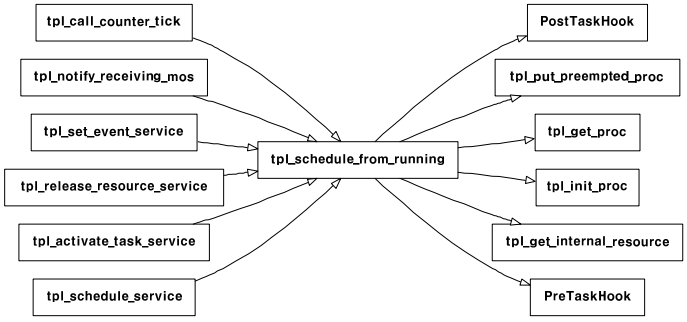
4. 적용결과

4-1. 적용 시스템 : Trampoline

검증대상인 Trampoline[4]은 국제표준(ISO 17356)

'OSEK/VDX'[5] 2.2.3 기반의 차량 전장용 운영체제로 ANSI C 로 구현된 오픈 소스 실시간 운영체제이다. ARM, POSIX, PPC, AVR, HCS12, C166 등 다양한 플랫폼에서 동작가능하며, POSIX 를 지원하기 때문에 Unix/Linux 환경에서도 동작할 수 있다. 검증대상의 측면에서 Trampoline 은 ANSI C 로 구현되어 있기 때문에 추상화 수준이 낮고, 절차지향적 프로그램이기 때문에 컴포넌트화 시키기가 힘들다는 단점이 있다.

4-2. 실험결과



(그림 5) tpl_schedule_from_running 의 의존성 그래프 (dependency depth = 1)

처음 검증 대상인 tpl_schedule_from_running(그림 5)는 태스크(task)가 러닝(running)상태이면서 스케줄링이 필요할 때 요청되는 함수이다. 다음 검증 대상인 tpl_put_new_proc 은 프로세스(process) 테이블에 수면상태인 프로세스나 새로운 프로세스를 넣는 함수이다.

<표 1>은 tpl_schedule_from_running, tpl_put_new_proc 와 연관성이 있는 개체들을 추출한 소스, 원본 소스의 코드 메트릭(code matric)을 비교한 것이다.

<표 1> 원본 소스와 추출 소스의 코드 메트릭 비교

메트릭	원본코드	추출 1	추출 2
코드 라인의 수	23,573	230	135
정의 문장 수	21,468	93	63
실행 문장 수	1,080	47	9
함수의 개수	266	7	1
사이클로메틱 복잡도의 합	494	15	3

*추출 1: tpl_schedule_from_running, *추출 2: tpl_put_new_proc

<표 1>과 같이 모든 수치가 적게는 95%에서 많게는 99%까지 감소했다. 특히 상태의 개수와 밀접한 연관이 있는 사이클로메틱 복잡도의 합(Sum Cyclomatic Complexity)이 tpl_schedule_from_running 에서는 494 개에서 15 개로, tpl_put_new_proc 에서는 3 개로 상당량 감소했음을 볼 수 있다. 그리고 정의되는 코드라인이 각각 21,468 개에서 93 개로, 21,468 개에서 63 개로 감소했으며 분석해야 될 함수의 개수도 266 개에서 7 개와 1 개로 대폭 감소했음을 볼 수 있다.

다음으로 실제 검증환경에서의 효율성에 대한 비교이다. 검증은 CBMC 를 통해 이루어졌으며 <표 2>는 이에 대한 결과이다.

<표 2> 원본 소스와 추출 소스의 검증 결과

	원본코드	추출코드 1	추출코드 2
할당식	2312	7	44
VCC	195	20	12
변수	3720046	476	1231
절(Clauses)	17633722	475	2371
경고(Claim)	273	14	12
시간(s)	213.818	0.006	0.194

*추출 1: tpl_schedule_from_running, *추출 2: tpl_put_new_proc

오류 검증대상의 크기가 줄어들기 때문에 <표 2>와 같이 모든 수치에서 적게는 90%, 많게는 99%의 감소 효과를 관찰할 수 있다. 특히 CBMC 는 코드의 모든 논리식들을 결합 정규형(CNF)을 기본으로 한 VCC(Verification Conditions)로 바꾸고 이를 SAT slover 로 풀어내는 방법을 통해 검증을 수행하기 때문에[6] 검증에 소요되는 자원의 면에 있어서 VCC 개수의 감소는 상당히 고무적이다. 수치상에 있어 VCC 의 개수는 tpl_schedule_from_running 의 검증에서 195 개에서 20 개로 대략 90%, tpl_put_new_proc 은 195 개에서 12 개로 대략 94% 가까이 줄어들었고, 실제 검증을 수행하는데 걸린 시간의 면에서 213.818 초가 걸리던 작업이 tpl_schedule_from_running 의 검증에서는 0.006 초로, tpl_put_new_proc 은 0.194 초로 대폭 감소함을 볼 수 있어 본 특성기반 코드 추출이 실제로 효과가 있음을 입증한다.

5. 결론

정형검증기법의 근원적인 문제인 상태폭발로 인한 자원의 과다소요는 이 기법을 효과적으로 적용하는 데에 가장 큰 걸림돌이 되어왔다. 본 연구에서 제시한 특성기반 코드추출기는 이러한 근원적인 문제를 해결하였다기 보다는, 문제를 발생시키는 원인을 축소하여 정형기법의 적용을 보다 현실화하였다는 데에 의미가 있다. 수작업으로 코드를 분석하고 추출하는 번거로움을 제거하여 정형기법의 적용프로세스를 간략화하고 검증자원을 효과적으로 절약하였다.

* 본 연구는 한국전자통신연구원의 위탁과제 "차량전장용 실시간운영체제의 안전성 검증기법 연구" 에 의해 지원되었습니다.

참고문헌

[1] Edmund M. Clarke and Jeannette Wing and et al. "Formal Methods: State of the Art and Future Directions" In *ACM Computing Surveys vol 28-4*, pages 626-643, 1996

[2] Gerard J. Holzmann. "*The SPIN Model Checker : Primer and Reference Manual*", Addison-Wesley Publishing Company, 2003

[3] Scitools. "http://www.scitools.com"

[4] Trampoline. "http://trampoline.rts-software.org"

[5] OSEK VDX. "http://www.osek-vdx.org/"

[6] Clarke, Edmund and Kroening, Daniel. "Hardware Verification using ANSI-C Programs as a Reference" In *Proceedings of ASP-DAC 2003*, pages 308-311, 2003