메모리 모델 적용을 통한 심볼릭 포인터 참조의 구현†

(The Implementation of Symbolic Pointer Dereference by Adapting of a Memory Model)

이 지 훈 ‡ 김 윤 호 § 김 문 주 ¶
(Jihun Lee) (Yunho Kim) (Moonzoo Kim)

요 약 소프트웨어의 신뢰성은 현대 IT 비즈니스에서 중요한 요소로 드러나고 있다. 현재 소프트웨어 검증 방법 중 가장 널리 쓰이는 테스팅 기법 성패는 효과적인 테스트 케이스의 생성 여부에 좌우된다. 2005년 동적 심볼릭 수행을 활용한 테스트 자동 기법이 제안된 이래 여러 연구팀에서 이 기법을 활용한 도구를 개발하였다. CREST는 UC Berkley 대학의 K. Sen 교수 팀에서 개발한 C언어를 위한 open source 테스팅 도구이다. 하지만, CREST에는 심볼릭 포인터 참조를 지원하기 위한 메모리 모델이 구현되어 있지 않아, 테스팅의 높은 커버리지를 얻기 어려운 측면이 있다. 따라서 본 연구에서는 CREST에 메모리 모델을 추가함으로써 심볼릭 포인터 참조를 구현하고 테스팅의 더 높은 커버리지를 얻고자 한다.

키워드 : 심볼릭 수행, 소프트웨어 테스팅, 심볼릭 포인터 참조, CREST

Abstract The reliability of software is becoming a critical issue of modern IT business. The success of software testing, which is one of the most popular software verification technique, depends on the ability to create test cases effectively. Since dynamic test generation technique is suggested in 2005, many research team worked on development of software testing tool with this technique. CREST is an open source testing tool which is developed by resarch tem of prof. Koshik Sen in University of California, Berkely. However, since a proper memory model to support symbolic pointer dereference is not implemented in CREST, we cannot get high coverage from testing of softwares contained symbolic pointer dereference. Thus, we want to add a proper memory model to support symbolic memory dereference which will be resulted in higher coverage of software testing.

Key words : Symbolic Execution, Software Testing, Symbolic Pointer Dereference, CREST

1. 서 론

소프트웨어의 신뢰성은 현대 IT 비즈니스에서 중요한 요소로 부각되고 있다. 현대 사회가 ubiquitous 컴퓨팅 사회로 발전함에 따라 현대인이 일상에서 사용하는 소형 기기, 가전기구에서부터 의료 기구, 통신 시스템, 우주산업에 이르기 까지 현대인의 생활 전반에 소프트웨어에 의해 구동되고 제어되고 있다. 따라서, 소프트웨어의 작은 오류가 현대 사회에 미치는 영향은 점점 확대되어 가고 있고, 다양해지고 있다. 예를 들어, 2009년 8월 대한민국 고유의 기술로 제작한 최초의 우주로켓 '나로호'가 소프트웨어 결함[1]으로 인해 발사를 8분가량 앞두고 발사를 멈추었다. 이처럼 소프트웨어 신뢰성은 날로 그 중요성을 더해가고 있고, 그 신뢰성을 검증하기 위한 효과적인 기술의 개발이 중요하다.

현재 소프트웨어 검증 방법 중 가장 널리 쓰이는 테스팅 기법 성패는 효과적인 테스트 케이스 생성 여부에 좌우된다. 그 중 동적 테스트 생성 기법(dynamic test generation)[2-8]은 대상 프로그램의 실제 수행을 통해 정적 테스트 생성 기법이 갖는 한계를 극복한다. 동적 테스트 생성 기법은 정적 심볼릭 수행으로 분석하기 어려운 구문이 있는 경우 실제 프로그램 수행에서 가지는 값을 활용하여 심볼릭 수행의 정확도를 최대한으로 높일 수 있다.

† 본 연구는 2010년도 한국과학기술원 학부생 연구 참여 프로그램(URP)의 연구비 지원으로 수행하였습니다.
‡ 비 회 원 : 한국과학기술원 전산학과
leejihun@kaist.ac.kr
§ 학생회원 : 한국과학기술원 전산학과
kimyunho@kaist.ac.kr
¶ 종신회원 : 한국과학기술원 전산학과 교수
moonzoo@cs.kaist.ac.kr
논문접수 :
심사완료 :

CREST[5]는 UC Berkley 대학의 K. Sen 교수 팀에서 개발한 C언어를 위한 open source 테스팅 도구로써, 동적 테스트 생성 기법을 사용하고 있다. 하지만, CREST는 심볼릭 메모리 구조를 지원하지 않기 때문에 심볼릭 포인터 참조(symbolic pointer dereference)가 지원되지 않고 있다.

CREST는 대상 프로그램(target program)을 C Intermediate Language(CIL)[9]를 이용하여 수정(instrumentation)하여 CREST가 생성한 입력에 따라 프로그램을 수행시키면서 프로그램 실행 경로가 어떤 분기를 따라가는지 기록한 심볼릭 경로 조건식(symbolic path constraint)을 생성한다. 이때, 대상 프로그램의 구문의 제약조건에 대하여 심볼릭 표현식을 포함하는 심볼릭 제약 조건을 생성할 수 없는 경우 프로그램의 실제 값을 이용하여 심볼릭 제약조건을 생성한다. 심볼릭 포인터 참조는 CREST가 수행할 수 없는 C의 표현식 이다. 이때, 심볼릭 포인터는 포인터가 참조하는 주소가 심볼릭 입력에 종속된 경우를 말하는데 심볼릭 포인터에 대한 참조를 심볼릭 포인터 참조연산이라 정의하고, C 표현식에 심볼릭 포인터 참조가 일어나는 해당 식을 심볼릭 포인터 참조식이라 얘기한다.

심볼릭 표현식의 실제화(concretization)는 대상 프로그램의 분기들과 경로들을 탐색하는 것에 실패하는 결과를 가져온다. 그 탓에 테스팅 도구가 프로그램에 내재한 결함을 찾아낼 수 있는 적절한 테스트 케이스의 생성을 불가능하게 할 수 있다.

심볼릭 제약조건의 실제화가 대상 프로그램의 검증에 미칠 수 있는 영향을 실제 예시를 통하여 알아보자.

```
01: void single_array(char x, char y) {
02:   char a[4];
03:   a[0] = x;
04:   a[1] = 0;
05:   a[2] = 1;
06:   a[3] = 2;
07:
08:   if (a[x] == a[y] + 2)
09:     assert( 0 );
10: }
```

그림 1 심볼릭 포인터 참조의 예시

그림 1의 함수 single_array는 입력 x, y를 입력받아 character 배열 *a*를 참조한다. CREST를 이용하여 이 함수를 검증할 때, 최초의 실행을 *{x=0, y=1}*의 입력과 함께 시작한다고 가정하자. 그러면 CIL모듈은 줄 8의 제약조건(constraint)을 심볼릭하게 해석하려 시도할 것이다. 만약 CREST가 줄 8의 조건식을 심볼릭하게 해석하는데 성공하면, 심볼릭 입력 *x, y*에 해당하는 메모리 *&a[x], &a[y]*에 저장된 심볼릭 값을 사용하여 해당 조건식을 비교할 것이다. 하지만, CREST에는 이를 지원하는 메모리 모델이 구현되어 있지 않으므로, 해당 조건식은 실제화되어 *{x == 2}*로 간단화(simplified)된다. 이 경우, 조건식이 거짓이므로, 줄 9가 실행되지 않고 프로그램이 종료된다. CREST는 다음 실행에서 줄 9로 분기하기 위하여 줄 8의 조건식을 참으로 만들기 위한 입력을 SMT solver[10]를 이용하여 생성할 것이다. 줄 8의 조건식은 *{x == 2}*로 간단화 되었으므로 가능한 다음 입력은 *{x= 2, y=1}*이 될 수 있고, 이 입력은 줄 9를 실행해야 한다. 하지만, *{x=2, y=1}*의 입력은 *{x=0, y=1}*로 대상 프로그램을 실행하였을 때와 같은 경로를 가지게 된다. 이는 도달 가능한(reachable) 모든 프로그램 구문을 테스팅을 통해 도달할 수 없게 되고, 과잉의 테스트 케이스의 생성을 가져오게 된다.

이 논문은 CREST가 심볼릭 포인터 참조를 지원하도록 적합한 메모리 모델을 추가하는 방안에 대하여 설명하고자 한다. 2장에서는 동적 테스트 생성 기법의 기초와 CREST의 구조를 설명한다. 3장에서는 새롭게 추가될 메모리 모델에 대해 설명하고, 4장에서는 3장에서 설명한 메모리 모델이 실제로 CREST에 적용된 방법에 대하여 기술한다. 5장에서 결론을 맺는다.

2. 연구 배경

2.1 동적 테스트 생성 기법

동적 테스트 생성 기법[2-8]은 주어진 입력으로 검증할 실제 대상 프로그램을 수행하면서 실행된 프로그램 경로를 따라 경로 제약 조건을 생성한다. 대상 프로그램 수행이 종료되고, 프로그램 수행 경로를 따라 생성된 경로 제약 조건을 이용하여, 이전에 수행되었던 경로와는 다른 경로를 수행하는 새로운 입력을 생성한다. 이 과정을 모든 수행 가능한 모든 프로그램 수행 경로를 수행할 때까지 반복하거나 사용자가 지정한 특정한 조건을 만족할 때까지 반복한다.

2.2 CREST

CREST는 UC Berkley 대학의 K. Sen 교수 팀에서 개발한 C 언어를 위한 open source 테스팅 도구이다. CREST는 C 언어로 작성된 프로그램 소스코드를 입력으로 받아 수행 가능한 모든 실행 경로를 탐색하는 테스트 케이스를 생성한다. 먼저, 프로그램 상태 변화를 기록할 수 있도록 CIL 소스 변환 도구를 사용하여 검증 대상 프로그램을 수정(instrumentation)하고 수정하는 대상 프로그램을 CREST가 생성한 임의의 입력에 따라 프로그램을 수행시키면서 프로그램 실행 경로가 어떤 분기를 따라가는지 기록한 심볼릭 경로 조건식을 생성한다. 다음 실행을 수행하기 전, 이전 프로그램 실행 경로와 다른 실행 경로를 탐색하기 위하여 이전 경로와 다른 분기를 따

라가게 될 새 경로 조건식을 만들고 선형 정수 조건식을 지원하는 SMT solver[10]를 사용하여 새 조건식을 따라가게 할 테스트 입력을 생성한다.

CIL을 이용하여 소스 코드를 수정할 때, 대상 프로그램의 구문마다 심볼릭 변수가 어떻게 바뀌고 현재 프로그램 실행이 어떤 실행경로를 따라가는지 기록하기 위한 probe를 삽입한다. 수정된 대상 프로그램은 CREST가 지정한 심볼릭 입력 값에 따라 실제로 프로그램을 수행하며 현재 실행 경로에 해당하는 심볼릭 경로 조건식을 기록한다. 대상 프로그램의 실행이 종료되면, CREST에서 기록된 심볼릭 경로 조건식을 활용하여 다음으로 수행될 프로그램 실행의 입력 값을 계산한다. 첫 번째 실행의 입력 값은 선언된 심볼릭 입력 변수 타입의 기본 값을 가지고, 다음 실행부터는 CREST에 의해 실행된 SMT solver에서 생성된 입력 값을 가진다.

수정된 대상 프로그램이 실행되면서 심볼릭 수행을 통해 심볼릭 경로 조건식을 생성한다. 각 심볼릭 변수는 심볼릭 처리기(symbolic interpreter)에 포함된 심볼릭 맵에 의해 관리되고, 각 할당 문(assignment sentence)이 실행될 때마다 할당 문의 심볼릭 변수에 해당하는 심볼릭 표현식을 갱신(update)한다. 프로그램이 조건 분기 문(branch statement)을 실행하게 되면 해당 조건 분기의 조건식을 심볼릭 표현식으로 나타낸 심볼릭 경로 조건식을 계산한다. 대상 프로그램의 실행이 종료되면 프로그램 실행중에 계산한 심볼릭 경로 조건식을 전부 결합한, 실행경로에 대한 심볼릭 경로 조건식이 생성된다.

대상 프로그램의 실행에서 생성된 심볼릭 경로 조건식은 다음 실행의 입력 값을 생성하기 위하여 사용된다. CREST는 심볼릭 경로 조건식에서 심볼릭 제약 조건 하나를 선택하여 부정하고(negate) 부정한 제약 조건 이후의 제약 조건을 지워 새로운 심볼릭 경로 조건식을 생성한다. 생성된 심볼릭 경로 조건식은 SMT solver 로 보내져 풀어지고, 구해진 해는 다음 프로그램 실행의 입력으로 쓰이고, 이때 수행되는 프로그램 경로는 이전 프로그램 실행 경로와는 다른 경로를 실행하게 된다.

CREST는 여러 가지 경로 탐색 전략을 포함하고 있는데, 사용자가 선택한 전략에 따라 가능한 모든 프로그램 실행 경로를 수행하거나 사용자가 지정한 제한 값에 도달할 때까지 프로그램 실행과 다음 입력 값 생성을 반복한다.

CREST는 심볼릭 경로 조건식으로 선형 정수 표현식으로 표현된다. 선형 정수 표현식에서 각 변수와 상수의 타입은 크기 제한이 없는 정수이고 각 표현식은 일차 식의 형태를 보인다. 따라서 사칙 연산 중 변수와 변수의 곱셈연산이나 나눗셈 연산은 지원되지 않고, C에서 지원하는 bit-wise 연산(&, |, ^ 등)을 지원하지 않는다. CREST가 검증 대상 프로그램의 C 표현식에 대하여 심볼릭 경로 조건식을 생성할 때, 선형 정수 표현식으로 표현할 수 없는 C 표현식을 해당 표현식을 실제화(concretization)하여 실행 당시에 해당 C 표현식이 가지는 변숫값을 사용하여 선형 정수 표현식으로 간소화한다. 본 연구의 심볼릭 포인터 참조 또한, 선형 정수 표현식으로 표시 될 수 없어서 해당 조건식은 실제화된 값으로 생성된 선형 정수 표현식으로 간단화 되게 된다.

3. 메모리 모델

심볼릭 포인터는 포인터가 참조하는 주소가 실제 프로그램 값이 아닌 심볼릭 입력에 종속된 포인터이다. 심볼릭 포인터 참조는 심볼릭 포인터에 대한 참조 연산(dereference)가 일어날 경우를 말한다. 심볼릭 포인터 참조는 대상 프로그램의 할당문의 오른쪽이나 조건식에 나타나는 심볼릭 읽기 연산이나 프로그램 할당문의 왼쪽에 나타나는 심볼릭 쓰기 연산이 있다. 본 논문에서는 이러한 심볼릭 포인터 참조의 기술을 위해 e가 심볼릭 표현식일 때, $*e$를 심볼릭 포인터 참조를 위한 표현식으로 사용 한다.

본 연구에서 사용된 메모리 모델[11]은 심볼릭 포인터가 참조할 수 있는 범위를 현실적으로 제한함으로써, 심볼릭 포인터 참조를 실용적으로 수행한다. 이론적으로 하나의 심볼릭 포인터 참조식은 32bit 주소 시스템을 가진 아키텍처에서 232가지 다른 실제 주솟값을 가질 수 있다. 하지만, 가능한 모든 경우의 주소를 참조하는 것은 매우 큰 비용을 발생시키고, 그 효용성이 떨어진다. 실제 대상 프로그램의 실행에서 포인터 참조식 $*e$는 유효한 메모리 영역의 한 위치를 가리켜야 한다. 이때, 유효한 메모리 영역은 그 영역의 시작 주소(start), 영역의 크기(size)로 정의될 수 있다. 본 메모리 모델에서 유효한 메모리 영역의 시작 주소와 그 크기는 심볼릭 입력에 종속되지 않는다고 가정한다.

심볼릭 포인터는 대상 프로그램의 실행 시 실제 메모리를 가리키는 실제화된 메모리 주소(concretized memory address)를 포함하는 유효한 메모리 영역의 집합을 정의한다. 그리고 실제화된 메모리 주소를 포함하는 유효 메모리 영역의 집합을 해당 심볼릭 주소가 참조하는 유효 메모리 영역(validmemoryregions)이라 정의한다.

심볼릭 포인터가 앞서 정의한 유효 메모리 영역의 주소만 참조하게 하는 것은 심볼릭 경로 조건식에 심볼릭 포인터가 유효 메모리 영역의 $start$와 $start + size - 1$ 사이에 있도록 제약 조건을 추가함으로써 이뤄질 수 있다. 또한, 유효 메모리 영역에 포함된 심볼릭 값들을 동적 테스팅 도구의 심볼릭 맵에 추가하고 심볼릭 포인터 참조가 일어나는 순간의 유효 메모리 영역 스냅 샷(valid memory region snapshot)을 심볼릭 포인터 참조식에 연관시켜 심볼릭 메모리 참조를 구현할 수 있다.

이 메모리 모델은 유효 메모리 영역의 시작 주소와 크기가 입력에 종속되지 않았을 때, 가장 정확한 메모리 모

델이다[11]. 따라서 본 모델을 이용하여 실용적이면서 정확한 심볼릭 포인터 참조를 구현할 수 있다.

3.1 메모리 모델 예시

그림 1의 single_array 함수에 본 메모리 모델을 적용해 보자.

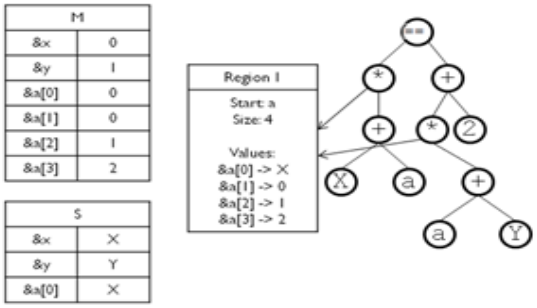

그림 2 그림 1의 single_array 함수의 표현식 나무(expression tree). (*) 연산자는 심볼릭 포인터 참조를 나타내고, 화살표는 심볼릭 포인터 참조가 일어날 당시의 유효 메모리 영역 스냅 샷(valid memory region snapshot)을 가리킨다.

그림 2는 그림 1의 single_array 함수가 {x=0, y=1}의 입력으로 시작되었을 때 생성되는 표현식 나무(expresion tree)를 나타낸다. 그림 1의 8번 줄의 조건식은 **(a + x) == *(a + y) + 2*로 다시 쓰일 수 있고 그에 따른 표현식 나무는 위와 같다. 이때, (*)연산자는 심볼릭 포인터 참조의 결과 값을 의미하고, 각 심볼릭 포인터 참조 노드(Node)에 연결된 표는 포인터 참조가 일어날 당시의 유효 메모리 영역의 스냅 샷을 나타낸다.

위의 예시와 같이 유효 메모리 영역 스냅 샷은 시작 주소(start address), 요소의 개수(size), 그리고 영역 내 포함된 실제 주소와 값의 매핑으로 이루어진다. 표 M과 표 S는 유효 메모리 영역 스냅 샷을 얻은 당시의 실제 메모리(concrete memory)와 심볼릭 메모리(symbolic memory) 상태를 나타낸다. 유효 메모리 영역 스냅 샷을 구할 때, 주소에 해당하는 매핑이 표 S에 존재하면 그 값을 가져오고, 그렇지 않을 때 표 M에서의 값을 가져온다.

4. 메모리 모델의 CREST에의 적용

CREST는 대상 프로그램의 심볼릭 수행을 위하여 심볼릭 처리기(symbolic interpreter)에 심볼릭 스택, 심볼릭 메모리 맵, 심볼릭 조건 절(symbolic predicate) 레지스터를 포함하고 있다. CIL에 의해 수정된 대상 프로그램이 실행되면서 심볼릭 처리기에 의해 동시에 심볼릭 수행이 이루어지고, 심볼릭 수행 후 기록된 심볼릭 경로 조건식을 이용하여 다음 입력을 생성한다.

기존 CREST는 심볼릭 포인터 참조는 심볼릭 수행을 진행하지 않고, 심볼릭 경로 조건식에 심볼릭 포인터 참조식이 포함될 경우, 실제화된 값을 이용하여 심볼릭 경로 조건식을 생성한다. 따라서, 심볼릭 포인터 참조를 심볼릭하게 수행하려면 CIL module을 수정하여 CREST의 심볼릭 수행 대상을 심볼릭 포인터 참조까지 늘려야 한다. 또한, 심볼릭 포인터 연산을 지원하기 위한 타입 시스템이 추가되어야 하고, 심볼릭 메모리 맵을 관리하기 위해 메모리 할당 정보를 추적해야 한다. 또한, 기존에 선형 정수 표현식으로 설계된 CREST의 심볼릭 표현식을 확대하며 생성된 심볼릭 경로 조건식이 SMT solver에 의해 풀릴 수 있도록 심볼릭 포인터 참조를 SMT solver 표현식으로 변환해야 한다.

4.1 타입 시스템

CREST가 포인터 연산의 심볼릭 수행을 진행하기 위해서는 타입 시스템의 구현이 필요하다.

```
01: void cast(int x){
02:     char a[4];
03:     long b[4];
04:     if(a[x] == b[x])
05:         assert(0);
06: }
```

그림 3 타입 시스템의 필요성 예시

그림 3의 4번 줄의 조건식은 **(a + x) == *(b + 4 * x)*로 다시 쓸 수 있다(32-bit 시스템). 포인터 연산의 심볼릭 수행을 위해서는 CREST가 배열 *a*와 *b*의 타입을 직접 알아야 한다. 따라서, CREST의 각 심볼릭 표현식에 타입 항목이 추가되어야 하고 원시 타입들에 대한 크기와 범위의 정의를 내려야 한다. 또한, 기존의 심볼릭 연산 기능들이 표현식의 타입을 고려하여 재 정의 되어야 한다.

4.2 메모리 할당 추적

본 연구에서 사용하는 메모리 모델은 심볼릭 포인터가 참조할 수 있는 유효 메모리 공간을 정의한다. 이때 심볼릭 처리기는 심볼릭 포인터 참조가 일어나는 시점의 메모리 상태를 추적할 수 있어야 하는데 이는 메모리 할당(allocation) 정보와 해제(deallocation) 정보를 추적함으로써 알 수 있다. 메모리 할당 정보는 할당된 영역의 시작 주소(start), 할당된 영역에 포함된 요소의 개수(size), 단위 요소의 타입(type)으로 정의될 수 있다. 구조체는 구조체의 멤버 변수들에 대한 개별적 메모리 할당 정보

추가와 구조체 변수 자체의 할당 정보 추가를 통하여 메모리 할당 상태를 추적할 수 있다.

```
01: void allocation(){
02:   char a[4];
03:   int b;
04: }
```

그림 4 메모리 할당 정보 예시

그림 4에서 *a, b, c*의 변수가 등록 된 것을 알 수 있다. 이때, 메모리 할당 정보는 아래와 같이 심볼릭 처리기에 기록되게 된다.

시작 주소	요소 개수	단위 타입
&a	4	char
&b	1	int

표 3 그림 4의 메모리 할당 정보

4.3 CIL 모듈 수정

CREST는 CIL에 외부 모듈을 추가하여 대상 프로그램의 수정에 이용한다. 기존 CREST는 심볼릭 포인터 참조를 실제화시키므로 CIL 모듈은 심볼릭 포인터 참조에 대해 심볼릭 수행을 위한 수정을 가하지 않는다. 따라서 심볼릭 포인터 참조를 심볼릭 수행하기 위해서는 CREST의 CIL 모듈을 수정하여야 한다.

CIL 모듈의 수정은 두 부분에서 일어나는데 메모리 할당 정보 추적을 위한 수정과 심볼릭 포인터 참조, 연산을 위한 수정이 필요하다. 먼저, 메모리 할당 정보 추적을 위해서, 대상 프로그램이 메모리를 할당하거나 해제하는 부분에 메모리 추적 probe를 넣어준다. 이때, 추적되는 정보는 4.2에서와같이 그 메모리 할당의 시작 주소, 요소 개수, 단위 타입으로 이루어진다.

심볼릭 포인터 참조가 일어나는 부분 또한 심볼릭 수행을 위한 probe를 삽입해야 한다. 이때, 심볼릭 배열 참조인 경우, 이를 포인터 연산식으로 풀어주어야 한다. 예를 들어 *a[x]*와 같은 배열 참조식은 **(a + x)*와 같은 포인터 연산식에 대한 참조의 형태로 해석되어야 한다. 풀어진 연산식은 심볼릭 스택을 이용하여 심볼릭하게 수행할 수 있다.

심볼릭 쓰기 연산의 경우도 CIL에서 고려해 주어야 하는데, 만약 심볼릭 쓰기 연산이 이전 프로그램 구문에서 존재하였으면, 그 이후의 모든 읽기 연산이 심볼릭 읽기 연산으로 처리되어야 한다. CIL 수정이 일어나는 동안에는 이전의 프로그램 구문에서 심볼릭 쓰기 연산이 일어나는지 알 수 없다. 따라서, 모든 포인터 참조 연산을 심볼릭 읽기 연산으로 가정하고 CREST로 넘겨와 심볼릭 처리기에서 해당 연산이 심볼릭하게 수행되어야 하는지 판단해야 한다.

4.4 심볼릭 표현식

기존의 CREST는 심볼릭 표현식을 선형 정수 표현식의 형태로 나타낸다. 따라서, 심볼릭 포인터 참조식의 경우 기존 CREST의 심볼릭 표현식으로 나타내어질 수 없다. 또한, 기존 CREST의 경우 변수와 변수의 곱셈이나 나눗셈 등, 다양한 연산을 지원하지 않는다.

일반적으로 수학적인 표현식은 연산 기호와 표현식을 이용하여 귀납적으로 정의할 수 있는데, 마찬가지로 C언어의 표현식 또한 같은 방법으로 정의할 수 있다.

```
e  := {unary operator} e
   || e {binary operator} e
   || e {compare operator} e
   || *e
   || {C variable}
```

그림 5 심볼릭 표현식의 정의

C의 표현식은 그림 5와 같이 정의할 수 있다. 위에 따르면 표현식 *e*는 귀납적으로 단항 연산자, 이항 연산자, 비교 연산자와 표현식 *e*가 결합한 형태로 나타날 수 있고, 또는 포인터 참조식 **e*가 될 수 있다. 또한, 기본적으로 *e*는 C의 변수를 나타낼 수 있다.

따라서, 심볼릭 표현식을 귀납적으로 정의하기 위하여 심볼릭 표현식을 클래스의 상속을 통하여 정의한다. 심볼릭 표현식 클래스를 가상 클래스로 정의하고, 위 정의에 따라 하위클래스들을 정의하여 심볼릭 표현식을 정의할 수 있다.

심볼릭 포인터 참조식의 경우 다른 하위 클래스들과 상이한 처리가 있어야 하는데, 심볼릭 포인터 참조식 클래스의 경우 생성과 함께 유효 메모리 영역을 정의하여야 한다. 정의된 유효 메모리 영역은 심볼릭 제약 조건을 통해 정의될 수 있고, SMT solver 표현식으로 변환되어 심볼릭 경로 조건식에 포함되게 된다.

4.5 제약 조건의 추가

심볼릭 포인터 참조를 SMT solver에 적용할 때는 SMT solver의 배열 이론을 사용한다. CREST에서 사용하는 SMT solver yices[13]의 경우 배열을 지원하기 위해 해석 되지 않은 함수(uninterpreted function)을 사용한다. 먼저 함수 타입의 SMT 표현식을 정의하여 SMT solver 환경에 등록한다. 다음 심볼릭 포인터가 참조할 수 있는 유효 영역들의 시작 주소와 사이즈를 이용하여, 범위를 제한시켜 준다.

유효 메모리 영역 또한 SMT 조건식을 사용하여 정의되는데, 메모리 영역의 각 요소가 주소와 그 주소 안의 심볼릭 값의 매핑(mapping)으로 심볼릭 포인터 참조식을 나타내는 SMT 표현식에 등록된다. 유효 메모리 영역

스냅 샷은 대상 프로그램의 순간의 메모리 상태를 나타내는 것이기 때문에 전체 메모리에 대해 함수를 하나 정의해도 메모리 충돌(memory conflict)이 일어나지 않는다.

심볼릭 포인터 참조식 e에 대해 유효 메모리 영역의 집합 이 얻어졌을 경우, e는 실제로 위의 n 개의 메모리 영역 중 하나를 가리키게 된다. e가 유효 메모리 영역 중 하나를 가리키게 하는 것은 심볼릭 경로 조건식에 제약 조건을 다음의 조건을 삽입하여 이뤄진다.

$$\bigvee_i (e == e_i) \wedge (start_i \leq e < start_i + size_i)$$

이때, $start_i$와 $size_i$는 유효 메모리 영역 의 시작 주소와 크기를 나타낸다. $e_i == e$는 e가 R_i을 가리키 것에 대한 필요충분조건이다. 이때, e_i는 e를 심볼릭하게 계산하면서 얻을 수 있는데 e가 다중 참조식으로 구성되어 있을 때, 가장 마지막 참조를 제외하고 참조 연산을 수행했을 때 얻어진다.

4.6 예시

그림 2는 그림 1의 8번 줄에 해당하는 표현식 나무를 보인다. 4.4 절에서 정의한 심볼릭 표현식 또한 그림 2처럼 정의되는데, 이때 (==) 연산자가 있는 노드를 비교 표현식, (*) 연산자가 있는 노드를 참조 표현식, (+) 연산자가 있는 노드를 이항 표현식으로 생각 할 수 있다. 각 참조 표현식은 실행 당시의 유효 메모리 영역 스냅 샷을 가지고 있고, 이 경우 같은 스냅 샷을 가지고 있다.

그림 1의 8번 줄을 4.5 절에 따라 다음과 같은 제약 조건으로 나타낼 수 있다.

먼저, 주소와 심볼릭 값의 맵을 선언해야 한다. SMT solver에 따라 맵을 지원하는 방법이 다른데, 현재 CREST에서 사용하는 yices에서는 맵을 해석되지 않는 함수를 사용한다. 따라서 메모리 맵을 함수 f라 정의하자. f는 메모리 주소를 입력으로 받고 그에 대한 심볼릭 값을 리턴하는 함수이다.

유효 메모리 영역 스냅 샷을 정의하기 위해 두 가지 종류의 제약 조건의 추가가 필요하다. 먼저, 메모리 영역을 정의하기 위해 해당 메모리 영역의 값들과 그 주소를 f를 이용해 매핑시켜야 한다.

```
C1: f(&a + 0) = x
C2: f(&a + 1) = 0
C3: f(&a + 2) = 1
C4: f(&a + 3) = 2
```

또한, 해당 함수의 정의역을 제한해야 한다. 이 경우 심볼릭 포인터 참조식들의 주소는 a+ x 또는 a+ y가 된다. 따라서 4.4절을 따라 아래의 제약 조건을 추가할 수 있다.

```
C5: &a <= &a + x < &a + 4
C6: &a <= &a + y < &a + 4
```

위의 제약 조건 중 C1에서 C4는 a[x]와 a[y]가 가리키는 유효 메모리 영역을 제약 조건을 이용해 정의해 준 것이고, C5와 C6는 해당 참조 표현식이 C1에서 C4가 정의하는 유효 메모리 영역만을 참조하도록 제한한다.

위의 제한 조건들을 SMT solver에 추가하고, 그림 1의 줄 8의 조건식을 SMT solver에 추가함으로써 심볼릭 포인터 참조를 수행할 수 있다. 해당 조건식은 아래와 같다.

```
C7: f(a + x) == f(a + y) + 2
```

위의 조건을 추가한 후, SMT solver를 이용해 해당 조건식을 풀이하면, {x=3,y=1}의 결과를 얻을 수 있다. 구해진 입력을 그림 1에 적용하면 이전에는 커버할 수 없었던 줄9를 수행할 수 있고, 대상 프로그램의 검증을 성공적으로 마칠 수 있다.

5. 결론

본 연구는 open source 테스팅 도구인 CREST에 적용 가능한 메모리 모델을 구현함으로써 CREST가 적용될 수 있는 대상 프로그램의 범위를 확장하였다. 메모리 모델의 구현은 CREST가 검증 가능한 C의 표현식의 범위를 넓혀 테스팅 커버리지를 높이고 불필요한 테스트 케이스의 생성을 낮추어 프로그램 검증의 질을 높이는데 이바지하였다.

본 논문에서는 간단한 예시만을 사용하여 단일 포인터 참조(single pointer dereference)에 대해서만 설명하였지만, 사용된 메모리 모델[11]은 다중 포인터 참조(multiple pointer dereference)를 지원하여, 다차원 배열 참조 또한 지원하는 모델이다. 메모리 모델의 구현은 다중 포인터 참조를 지원하도록 구현되었다.

앞으로 추가로 진행될 연구에서 본 연구에서 개발된 개선된 CREST를 이용하여 메모리 모델의 추가를 통한 SMT solver의 배열 이론의 적용이 기존 선형 정수 이론과 비교하여 실제적으로 그 효율성을 증가시킬 수 있는지 사례 분석을 진행할 것이다.

참 고 문 헌

[1] 고홍, 「나로호, 자동 발사장치 소프트웨어 결함」, 『중앙

일보』, 2009.8.21

[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, 2005

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," International Symposium on the Foundations of Software Engineering, 2005

[4] CREST Project Page
http://code.google.com/p/crest

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing", Annual network and Distributed System Security Symposium, 2008

[6] N. Tillmann, and J. Halleux, "Pex-White Box Test Generation for .NET," International Conference on Tests and Proofs, 2008

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death", ACM Conference on Computer and Communications Security, 2006

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", USENIX Symposium on Operating System Design and Implementation, 2008

[9] G. C. Necula, S. McPeak, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," International Conference on Compiler Construction, 2002

[10] SMT-LIB: The Satisfiability Modulo Theories Library, http://combination.cs.uiowa.edu/smtlib/.

[11] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise Pointer Reasoning for Dynamic Test Generation", International Symposium on Software Testing and Analysis, 2009

[12] 김윤호, 김문주, 동적 심볼릭 수행을 응용한 테스트 케이스 자동 생성 도구 비교, Korea Conference on Software Engineering, 2010

[13] Yices: An SMT Solver
http://yices.csl.sri.com/

이 지 훈
2007년~현재 한국과학기술원 전산학과 학사과정. 관심분야는 소프트웨어 테스팅, 자연 언어 처리

김 윤 호
2007년 한국과학기술원 전산학과 (학사)
2007년~현재 한국과학기술원 전산학과 석/박사 통합과정. 관심분야는 임베디드 소프트웨어, 소프트웨어 모델 체킹, 소프트웨어 테스트 자동 생성

김 문 주
1995년 KAIST 전산학과 학사.
2001년 Univ. of Pennsylvania 박사.
2002년~2004년 SECUi.COM 차장.
2004년~2006년 POSTECH 연구원.
2006년~현재 KAIST 전산학과 조교수.
관심분야는 정형검증, 소프트웨어 테스팅, 내장형 소프트웨어