통합된 버그잡는 기술들 : SLAM과 Bandera

한국과학기술원 양홍석 · 이욱세*

1. 서 론

최근 소프트웨어의 신뢰성에 관심이 높아지면서 프로그램 검증 기술이 부각되고 있다. 운영체제의 오동작, 로켓 내 프로그램 오동작으로 인한 폭발, 의료기기의 오동작으로 인한 인명 손실 등 검증되지 않은 소프트웨어로부터의 손실을 경험하면서 신뢰성에 대한 관심이 높아졌다. 단지 테스트(test)만 이용해서는 신뢰성을 높이는데 한계가 있기 때문에, 소프트웨어가 언제나 의도한대로 작동한다고 100% 보장해 줄 수 있는 프로그램 검증 기술이 주목을 받았다. 특히, 근래에 나온 검증 기술에 대한 눈부신 연구 결과들이 검증 기술에 대한 편견을 깨뜨린 이후에 더욱 그러했다. 즉, 지금까지 검증 기술은 실제로 사용하기에 너무 많은 비용이 든다고 여겨졌으나, 이 분야에 최근 연구 결과에 따르면 검증 기술은 종종 테스트보다도 적은 비용이 들고 테스트에서 발견하기 힘든 버그들을 찾아준다고 밝혀졌다.

1990년대까지 프로그램 검증은 주로 사용하는 기술이 프로그램 분석(program analysis)[1,2]인가, 모델 검증(model checking)[3,4]인가 아니면 자동 증명 기술(theorem proving)[5,6]인가에 따라 나뉘어졌다. 프로그램 분석이란 주어진 프로그램을 요약하여 프로그램이 가지는 성질을 예측하는 기술이고, 모델 검증은 검증하려는 프로그램이 가질 수 있는 모든 상태를 효과적으로 검색하여 명세를 만족하지 않는 경우가 있는지 찾아내는 기술이다. 마지막으로 자동 증명 기술은 수학의 명제 중 참인 것을 자동으로 증명하는 기술이다.

최근 개발된 검증 기술들의 특징은 프로그램 분석, 모델 검증, 자동 증명 이 세 가지를 총동원하여 사용한다는 점이다. 세 가지 서로 다른 분야의 최신 기술을 결합함으로써 기존의 검증 기술로는 찾아내기 어려운 버그들을 잡아 낼 수 있었다. 이 글에서 우리는 세 가지 기술을 통합하여 사용하고 있는 대표적인 예인 Microsoft의 SLAM 프로젝트[7]와 Kansas 주립대학의 Bandera 프로젝트[8]를 소개하고자 한다. 또한, 이들은 실제 현장에서 성공적이었던 사례로 꼽히고 있다.

* 학생회원

2. 적용되고 있는 기술들

2.1 SLAM 프로젝트 [7]

SLAM은 Microsoft 연구소에서 프로그램이 원하는 속성을 만족하는지 검증하고 버그를 발견해 주어 신뢰성 높은 소프트웨어를 제작토록 도와주기 위한 프로젝트이다. 방법론은 앞서 이야기한 세 가지 기술을 규합하여 완전하지는 않지만 실용적으로 사용될 수 있는 도구를 만드는 것이다.

Microsoft는 SLAM 프로젝트의 결과물을 이용하여 Microsoft Windows XP가 사용하는 디바이스 드라이버(device driver)의 버그들을 찾아내고 있다. 윈도우즈의 많은 버그들 중에는 드라이버에서 발생하는 버그들이 많은데, 왜냐하면 이런 드라이버들은 새로운 기기가 추가될 때마다 새로이 작성되고 개발자들이 보통 운영체제를 개발한 쪽과 다르기 때문이다. SLAM의 결과물로 현재 100 여 개의 드라이버에 대해 20 여 개의 속성을 검증하여 20 개 이상의 버그를 찾아냈다고 한다.

SLAM 프로젝트가 취하고 있는 방법론의 핵심은 프로그램을 요약하여 검증하고, 검증에 실패하면 반례(counter-example)를 기반으로 요약된 프로그램에 필요한 부분을 추가하여 검증하는 것을 반복하는

것이다. 이런 과정 중에 프로그램 분석, 모델 검증, 자동 증명 기술 등이 녹아 있다.

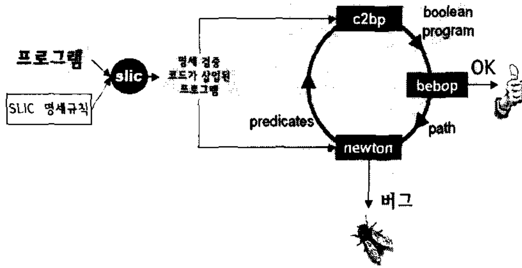

그림 1 SLAM의 핵심 과정

```
type enum {Locked, Unlocked};
s = Unlocked;
SlicAcquireLock() {
    if (s==Locked) abort;
    else s = Locked;
}
SlicReleaseLock() {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
do {
    SlicAcquireLock();
    AcquireLock();
    nOld = n;
    if(req){
        req = req->Next;
        SlicReleaseLock();
        ReleaseLock();
        n++;
    }
} while (n != nOld);
SlicReleaseLock();
ReleaseLock();
```

그림 2 명세 규칙과 검증 대상 프로그램

먼저 주어진 명세 규칙(specification rules)이 지켜지는지 검사하는 코드를 프로그램에 삽입한다. 예를 들면, 락(lock)은 걸려 있을 때만 풀 수 있고 풀려 있을 때만 걸 수 있다. 이를 명세 언어 SLIC[9]으로 작성하면, 프로그램에서 락을 걸고 푸는 부분에 락의 상태를 검사해서 규칙을 어기면 오류를 발생시키는 코드를 삽입해 준다. 예를 들어, 그림 2의 프로그램에서 이탤릭으로 쓰이지 않은 부분이 검증 대상 프로그램이고, 이탤릭으로 쓰여진 부분이 검증을 위해 삽입된 코드이다. 삽입된 코드는 락에 대한 상태 변수를 s라 하고, 락을 걸 때 (AcquireLock 호출 직전) 락이 이미 걸려 있으면 abort를 호출하고, 걸려 있지 않으면 상태를 Locked로 바꾼다. 같은 방법으로 락을 풀 때(ReleaseLock 호출 직전) 락이 이미 풀려 있으면 abort를 호출하고, 아니면 상태를 Unlocked로 바꾼다. 이렇게 변환한 후에는 명세 규칙을 준수하는 문제가 abort가 호출되느냐의 문제로 바뀐다.

```
p1 = false;
SlicAcquireLock() {
    if (p1) abort;
    else p1 = true;
}
SlicReleaseLock {
  if (!p1) abort;
  else p1 = false;
}
do {
    SlicAcquireLock();
    if(*){
        SlicReleaseLock();
    }
} while (*);
SlicReleaseLock();
```

그림 3 첫 번째 요약된 프로그램

c2bp[10] : 검증에 필요한 부분은 요약하고 나머지는 지운다. c2bp는 명세 검증 코드가 삽입된 프로그램과 명제들(predicates)들을 입력으로 받는다. 명제란 “x==3”과 같이 프로그램 실행 중에 참 또는 거짓을 값으로 가지는 문장이다. c2bp는 입력으로 주어진 명제들에 영향을 주는 부분은 요약하고 그렇지 않은 부분은 지운다. 처음으로 실행할 때에는 명세 규칙으로부터 명제들을 얻어 c2bp에 입력으로 주고, 그렇지 않은 경우에는 바로 전에 c2bp를 실행할 때 입력으로 주었던 명제들과 newton이 추가로 제공되는 명제들을 합쳐 입력으로 준다. 예에서는 락의 상태에 대한 명제 “s==Locked”(p1이라 하자)를 입력으로 한다. 결과적으로 락의 상태를 저장하는 변수 s를 검사하거나 바꾸는 부분은 요약하고 나머지는 지운다. 그림 3은 변환한 후 예제 프로그램을 보여준다. 변환된 프로그램에서 p1은 “s==Locked”인지를 나타내는 논리값 변수로, if의 조건으로 쓰인 “p1”은 “s==Locked”, “!p1”은 “s==Unlocked”를 나타낸다. 또한 “p1=true” 는 “s=Locked”를, “p1=false” 는 “s=Unlocked”를 나타낸다. while이나 if의 조건으로 *가 쓰이기도 하는데 이것은 true 또는 false 어느 쪽도 가능하다는 것을 나타내며, 조건이 p1과는

상관이 없을 때 사용된다.

bebop [11] : 요약된 코드가 명세 규칙을 만족하는지 모델 검증 기술로 검증한다. 여기서 프로그램의 제어 흐름을 알기 위해 프로그램 분석을 사용한다. 검증의 답은 두 가지로 나오는데, "yes"는 명세 규칙을 만족한다는 것이므로 검증 끝낸다. "no"는 규칙을 만족하지 않을 수 있다는 것으로 다음 과정으로 진행한다. "no"라는 답이 나왔다고 해서 프로그램이 반드시 규칙을 어기지는 않는데 이는 c2bp가 프로그램을 요약하는 과정에서 안전하지만 정확하지 않은 추정(approximation)을 했기 때문이다. 예제 프로그램에 대해 bebop은 "no"라고 답을 하는데, 왜냐하면 if문 내의 SlicReleaseLock을 호출하고 while문을 벗어난 후 SlicReleaseLock을 호출할 수 있기 때문이다.

newton [12] : 검증이 실패하면 정말로 오류가 있는 것인지 확인한다. 문제 소지가 있는 실행 경로(path)를 bebop이 넘겨 주면 실제로 그 경로를 따라 실행할 수 있는지 검사한다. 이 때 자동 정리 증명기(theorem prover)를 사용한다. 답은 세 가지가 나올 수 있는데, "yes"가 나오면 버그를 찾은 것이고, "no"가 나오면 이 경로는 실행 불가능하다고 알아낸 것이다. 이 경우에 newton은 이 불가능한 경로를 제외시키면서 검증할 수 있도록 정보를 제공한다. "maybe"가 나올 수도 있는데, 이는 검증이 불가능하다는 것으로 사용자의 입력이 요구된다. 예제에 대해 newton은 "no"라고 대답하는데, 왜냐하면 if문의 조건이 만족하여 "n++"이 실행되면 "n!=nOld"가 성립하므로 결코 while 루프를 벗어날 수 없기 때문이다. 추가로 제공되는 정보는 "n==nOld"라는 명제도 분석되어야 한다는 것이다.

newton이 "no"를 대답하면서 분석해야 할 명제를 제시하면 c2bp부터 다시 반복한다. 이 때, c2bp는 추가된 명제도 분석 가능하도록 프로그램을 요약한다. 예제 프로그램은 그림 4와 같이 요약되는데, 추가된 것은 "n==nOld"(p2라고 하자)에 관련된 부분이다. 첫 번째 "n = nOld" 는 "p2 = true"로 변환되고, 두 번째 "n++"은 p2가 참일 때는 거짓이 되고, p2가 거짓일 때는 결과를 알 수 없으므로 *을 준다. 마지막 while의 조건은 !p2로 변환된다. 이 변환 과정은 경우에 따라서 매우 복잡한 계산을 요하는데, SLAM에서는 자동 정리 증명기(theorem prover)를 사용하여 계산한다.

이러한 과정을 반복하면 분석해야 할 명제가 증가하게 되므로 요약된 프로그램은 점점 더 자세해 진다. 이러한 반복과정은 이론적으로는 끝나지 않을 수도 있는데, 실험에 의하면 만 줄 정도의 Windows XP 디바이스 드라이버에 대해서 20번 반복을 넘은 적이 없다고 한다. 예제 프로그램의 경우, 두 번째 검증 작업에서 "yes"가 나와 락(lock)의 규칙을 어기지 않음이 증명된다.

SLAM 프로젝트는 학계와 산업계에서 동시에 주목을 받고 있다. 학계에서는 버그 잡는 기술인 프로그램 분석, 모델 검증, 자동 증명 기술들을 통합하여 완전하진 않지만 정교하고 현실적인 결과물을 내고 있다고 보고 있다. 산업계에서는 정확하게 버그를 잡을 수 있고 많은 종류의 버그[1]를 한 방법론을 사용하여 검사할 수 있어서 주목하고 있다. 게다가 버그의 양산지인 Microsoft에서 버그에 관심을 갖고 연구개발에 투자해 얻어낸 결과물이고 향후 Microsoft의 제품에 적용될 가능성이 높다는 것에도 의미가 있다.

```
p1 = false;
SlicAcquireLock() {
    if (p1) abort;
    else p1 = true;
}
SlicReleaseLock {
  if (!p1) abort;
  else p1 = false;
}
do {
    SlicAcquireLock();
    p2 = true;
    if(*){
        SlicReleaseLock();
        p2 = p2 ? false : *;
    }
} while (!p2);
SlicReleaseLock();
```

그림 4 두 번째 요약된 프로그램

2.2 Bandera 프로젝트 [8]

Bandera는 Java로 작성된 프로그램이 올바르게 작동하는지를 검증해주는 프로그래밍 도구이다. 특히 여러 쓰레드(thread)가 동시에(concurrent) 수행

1) 정확히는 유한 시간적인 속성(temporal safety property)에 관한 버그. 예를 들어, 사용 순서 규칙이 있는 라이브러리를 사용할 때 그 규칙을 어기는 버그들. 많은 프로그램 버그가 이 범주안에 속한다.

되는 경우에 발생 가능한 오류를 찾아준다. 예를 들어, 모든 쓰레드가 서로를 기다리기만 하는 교착 상태(dead lock)에 빠지지 않는지, 또는 무한정 어떤 조건이 만족되기를 기다리기만 하는 쓰레드가 없는지 검사해 준다.

Bandera는 Kansas 주립 대학에서 1998년부터 개발해왔고 실제 현장에 적용되고 있다. NASA와 Honeywell technology center에서 Bandera를 사용하여 개발하는 소프트웨어를 검증하고 있다고 한다. 특히 Honeywell technology center는 Bandera를 이용하여 항공 운항을 위한 실시간 운영체제인 DEOS (Digital Engine Operating System)에서 오류를 찾아냈고 수정할 수 있었다고 한다[13].

Bandera의 검증 과정은 크게 세 가지로 나뉘어져 있다. (1) 먼저 Java 프로그램과 검증하려고 하는 명세(specification)를 잘 알려진 모델 검증기가 받아들이는 언어로 프로그램 분석 기술을 사용하여 변환한다. (2) 모델 검증기를 이용하여 변환된 프로그램이 변환된 명세를 만족하는지 검사한다. 검증에 성공하면 명세를 만족한다고 보고하고, 검증에 실패하면 만족하지 않는 예를 찾아낸다. (3) 검증에 실패했을 때, 찾아낸 오류의 예를 사용자에게 직접 보이지 않고, 오류를 역추적하여 Java 프로그램에 대한 예로 변환하여 보여준다. 이 과정은 그림 5에 나타나 있다.

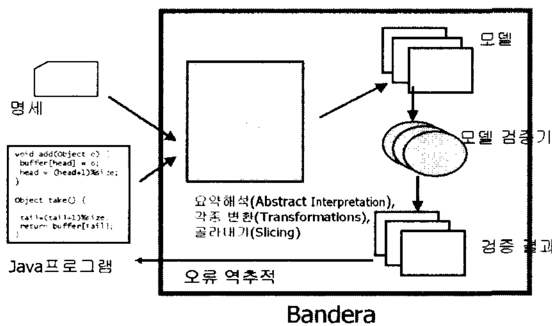

그림 5 Bandera의 검증 과정

프로그램 분석 기술을 사용하여 Bandera는 Java 프로그램과 그에 대한 명세를 요약하여 모델 검증기가 받아들이는 언어로 변환한다. 두 가지 기술이 사용되는데, 하나는 프로그램에서 부분만 골라내는 기술(program slicing)[14]이고, 다른 하나는 요약 해석 기술(abstract interpretation)[1,2]이다.

골라내기 기술(slicing)을 사용하여 검증하려고 하는 명세에 영향을 미치는 부분만을 골라낸다. 그림 6의 프로그램에서 BoundedBuffer라는 클래스(class)를 예로 들어 이 과정을 자세히 설명하겠다. BoundedBuffer 클래스는 한번 만들어지면 그 크기가 변하지 않고 여러 쓰레드가 공유해서 사용할 수 있는 버퍼(buffer)를 구현하고 있다. 이 BoundedBuffer 프로그램에 다음과 같은 명세를 생각하자: "어떤 순간에 버퍼가 비어있다고 가정하자 (head == (tail+1) % bound). 그 이후 어떤 순간에도 메쏘드(method) add를 완전히 (마지막 return명령어까지) 수행한 쓰레드가 하나도 없다면, 메쏘드 take를 완전히 수행한 쓰레드도 없다." 버퍼에 저장된 값은 위 명세를 검증하는데 영향을 미치지 않으므로 버퍼에 값을 저장하거나 읽는 부분을 제거한다. 명세와 관련 있는 것은 버퍼에 빈 공간이 얼마나 남아 있는가 하는 사실이다. 이 과정을 거치면 그림 7과 같은 프로그램으로 변환된다. 골라내기(slicing) 과정은 Bandera의 수행속도를 빠르게 한다. 만약, 주어진 프로그램의 많은 부분이 명세와 관련이 없어 사라진다면, 검증해야 할 프로그램은 작아진다. 따라서, 모델 검증기는 보다 효율적으로 검증을 할 수가 있다.

```
class BoundedBuffer {
   Objects [] buffer;
   int bound, head, tail;
   public BoundedBuffer(int b) {
      bound = b;
      buffer = new Object[bound];
      head = 0;
      tail = bound -1;
   }
   public synchronized void add(Object o) {
      while (tail == head )
         try { wait(); }
         catch (InterruptedException ex) {}
      buffer[head] = 0;
      head = (head + 1) % bound;
      notifiyAll();
   }
   public synchronized boolean isEmpty() {
      return head==(tail+1)%bound;
   }
   public synchronized Object take()
      while (isEmpty())
         try { wait(); }
         catch (InterruptedException ex) {}
      tail = (tail+1) % bound;
      notifyAll();
      return buffer[tail];
}
```

그림 6 Java로 작성된 BoundedBuffer 프로그램

Bandera는 요약 해석(abstract interpretation)을 사용하여 검색 공간을 유한하게 줄인다. 모델 검증기가 받아들이는 프로그램은 Java 프로그램과 큰 차이를 보인다. Java 프로그램에서는 컴퓨터가 무한히 많은 상태를 가질 수 있다고 가정하는 반면, 모델 검증기는 항상 유한한 상태만을 검증할 수 있다. 그러므로, 검증을 위해서는 유한한 상태만을 가질 수 있도록 프로그램을 변환해야 한다. BoundedBuffer 프로그램의 경우, 요약 해석 과정은 우선 bound, head, tail 그리고 메쏘드의 입력 값을 나타내는 b, o등이 가질 수 있는 값과 동적 메모리의 크기를 유한하게 제한한다. 예를 들어, bound, head, tail들에 저장되는 정수값은 {L,0,1,2,U}로 요약할 수 있다. L는 모든 음의 정수를 나타내며 U는 3이상의 모든 정수를 나타낸다. 그리고, 요약 해석은 bound, head, tail, b, o 또는 동적 메모리에 대한 모든 연산자를 요약된 값들에 대해 작용하도록 변환한다.

```
class BoundedBuffer {
    int bound, head, tail;
    public BoundedBuffer(int b) {
        bound = b;
        head = 0;
        tail = bound-1;
    }
    public synchronized void add(Object o) {
        while (tail == head )
            try { wait(); }
            catch (InterruptedException ex) {}
        head = (head + 1) % bound;
        notifiyAll();
    }
    public synchronized boolean isEmpty() {
        return head==(tail+1)%bound;
    }
    public synchronized Object take()
        while (isEmpty())
            try { wait(); }
            catch (InterruptedException ex) {}
        tail = (tail+1) % bound;
        notifyAll();
        return nil;
    }
}
```

그림 7 골라내기를 적용한 후의 BoundedBuffer 프로그램

기술적인 면에서 보았을 때 Bandera는 모델 검증기의 전처리기로 프로그램 분석을 사용한 것이다. 일반적으로 모델 검증기는 가능한 모든 상태를 추적해서 검증을 하기 때문에, 가능성이 유한하고 또한 적어야 효율적으로 검증할 수 있다. Bandera 개발진은 모델 검증기는 하드웨어 검증에서 탁월한 능력을 보여 준 기존의 것을 사용하고, 소프트웨어 검증을 위해 프로그램 분석 기술을 이용하여 프로그램을 요약하여 효율적인 검증이 가능하도록 하는데 초점을 두었다.

Bandera의 접근 방법은 SLAM과 같이 편리하고 정교하진 않지만, 검증할 수 있는 속성이 더 일반적이다. SLAM은 단순히 API 사용 패턴과 같은 유한 시간적인 속성(temporal safety property)만을 다루고 있는데 Bandera는 모든 시간적인 속성(temporal property)을 검증할 수 있다. 단지 정교한 검증을 위해서는 사용자가 적절한 요약 방법을 제시해야만 하는 불편이 있다.

3. 결 론

이 글에서는 프로그램 분석/검증 기술들을 통합하여 실제 현장에서 버그를 찾아내는데 노력한 두 가지 프로젝트 SLAM과 Bandera를 소개하였다. 물론, 이 글에서 소개하지 못한 많은 뛰어난 연구 결과들도 많다 [15,16,17]. 그러나, SLAM과 Bandera가 돋보이는 것은 프로그램 검증 분야에서 오랫동안 개발된 성숙된 연구 결과물을 총동원하여 실제 현장에서 사용할 수 있도록 실용적으로 개발하고 있다는 것이다.

버그 잡는 기술은 이렇게 성숙된 연구 개발물들이 가시화되고 있고, 많은 산업 분야에서 이용되고 있으며 앞으로도 더 많이 이용될 것으로 보인다. PC에서 돌아가는 응용 프로그램뿐만 아니라, 핸드폰, 가전기기, 자동차/항공기, 각종 기계에 들어가는 내장형 소프트웨어들, 특히, 안전성이 중요시되는 핵발전소, 자동차/항공기, 의료기기, 우주선 등에 내장되는 소프트웨어에도 버그들이 도사리고 있다. 많은 외국 기업들은 이미 이런 버그들에 의해 손실을 겪고 소프트웨어의 안전성에 깊은 관심을 보이고 있다. 이렇게 버그 잡는 기술에 대한 연구가 가시화되고 관심이 깊어지면서 점점 더 버그 잡는 기술에 대한 연구와 개발은 뜨거워 질 것으로 예상된다. 이론과 실제를 겸비한 연구가 필요한 때이다.

참고문헌

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In Proceedings of the ACM Symposium on Principals of Programming Languages, pages 238~252, January, 1977.

[2] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. Journal of Logic and Computation, 2(4):511~547, 1992.

[3] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool~supported program abstraction for finite~state verification. In Proceedings of International Conference on Software Engneering, pages 177~187, 2001.

[4] E. M. Clarke, O. Grumberg, and D. Peled. Model Checking, The MIT Press, 1999.

[5] G. Nelson. Techniques for Program Verification. Ph.D. thesis, Stanford University, 1980.

[6] D. Detlefs, R. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Research Report 159, Compaq System Research Center, December, 1998.

[7] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 1~3, January, 2002. http://research.microsoft.com/projects/slam.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite~state Models from Java Source Code. In Proceedings of the International Conference on Software Engi neering, June, 2000. http://www.cis.ksu.edu/santos/bandera.

[9] T. Ball and S. K. Rajamani. SLIC: A Specification Language for Interface Chec king (of C). Technical report MSR-TR -2001-21, Microsoft Research, 2001.

[10] T. Ball, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In Proceedings of Conference on Programming Language Design and Implementation, pages 203~213, 2001.

[11] T. Ball and S. K. Rajamani. Bebop: A Path-sensitive Interprocedural Dataflow Engine. In Proceedings of Workshop on Program Analysis for Software Tools and Engineering, pages 97~103, 2001.

[12] T. Ball and S. K. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical report MSR-TR-2002-09, Microsoft Resear ch, 2002.

[13] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS real-time Scheduling Kernel. In Proceedings of International Conference on Software Engineering. June, 2000.

[14] M. B. Dwyer, J. Hatcliff, and H. Zheng. Slicing Software for Model Construction. In Proceedings of the ACM SIGPLAN Partial Evaluation and Program Manipulation, January, 1999.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In Proceedings of Conference on Programming Language Design and Implementation, 2002.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code, Computer-Aided Verification, 2002.

[17] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - A second generation of a Java model checker. Workshop on Advances in Verification. July, 2000.

양 홍 석

1996 한국과학기술원 전산학과 학사
2001 일리노이 주립대학 전산학과 박사
2001~한국과학기술원 위촉연구원
관심분야 : 프로그램 분석, 프로그램 의미론, 프로그램 검증
E-mail : hyang@ropas.kaist.ac.kr

이 욱 세

1995 한국과학기술원 전산학과 학사
1997 한국과학기술원 전산학과 석사
1997~한국과학기술원 전자전산학과 박사과정
관심분야 : 프로그램 분석, 컴파일러, 프로그램 검증
E-mail : cookcu@ropas.kaist.ac.kr

• 제13회 통신정보 합동 학술대회 •

- 일 자 : 2003년 4월 30일~5월 2일
- 장 소 : 안면도 롯데오션캐슬
- 주 최 : 정보통신연구회 · 한국통신학회 · 대한전자공학회
  한국통신정보보호학회 · 한국정보처리학회
- 문 의 처 : 서울대 이광복 교수
  Tel. 02-880-8415
  http://jcci21.or.kr