Search-based JavaScript Modeling for Static Analysis

Group Report

1. Definition of Problem

자바스크립트가 소프트웨어 산업의 주요 언어로 떠오르면서, 소프트웨어의 안정성을 위해 자바스크립트 코드를 검사하는 시도 또한 많아지고 있다. 자바스크립트는 웹 어플리케이션을 구성하는 주요 요소일 뿐만 아니라, 어떤 기기이든 동작할 수 있어 모바일과 스마트 기기 어플리케이션 개발에도 사용되고 있다. 그러나 자바스크립트의 동적인 특성은 많고 다양한 소프트웨어 결함을 야기할 수 있어, 개발자가 안전한 프로그램을 개발하기 어렵게 만든다. 따라서, 프로그램의 안정성을 검사하는 다양한 연구가 진행되고 있다. 그 중 이번 그룹 프로젝트와 관련된 검사 방법은 정적 분석으로, 프로그램을 실행하지 않고 소프트웨어 결함을 검출한다. 그러나 자바스크립트의 동적인 특성과 다양한 라이브러리의 사용 등은 정적 분석을 이용해 결함을 검출하기 어렵게 만든다. 이러한 어려움 중 이번 그룹 프로젝트에선 “opaque code”로 인한 정적 분석의 어려움을 다룬다.

Opaque code란 실행 가능하나 소스 코드를 알아낼 수 없거나 처리하기 어려운 코드로, 정적 분석은 주로 모델을 만들어 이를 다룬다. 예를 들면, `Array.prototype.shift` 같은 자바스크립트의 다양한 built-in 함수, `document.getElementById` 같은 DOM 관련 함수, 또는 `Tizen.getDeviceId` 같은 특정 스마트 기기에서만 사용 가능한 함수가 opaque code 이다. 대부분의 정적 분석은 모델링을 통해 이러한 opaque code을 다루며, 모델링은 주로 수작업으로 만들어진다. 그러나 수작

업으로 모델을 만드는 건 opaque code 마다 모델을 일일이 만들어야 하므로 시간이 오래걸리며 opaque code가 복잡할 경우 실수하기 쉽다.

따라서 이번 그룹 프로젝트에선 search-based approach를 이용하여 opaque code에 대한 모델을 자동으로 생성하려 한다. 여기서 모델은 opaque code의 행동을 나타낼 수 있는 자바스크립트 프로그램이다. 이 프로젝트는 International Symposium on the Foundations of Software Engineering (FSE) 학회에서 작년에 발표한 자동 모델링 도구 `Mimic`에 기반을 두고 있고, 다양한 heuristics을 이용해 `Mimic` 보다 빠르게 모델을 생성하는 걸 목표로 하고 있다.

2. Background (`Mimic`)

이번 그룹 프로젝트의 기반인 `Mimic`은 다양한 입력 값에 대해 opaque code의 execution trace를 기록하여, 이를 토대로 모델을 (자바스크립트 프로그램을) 생성한다. 사용자가 초기 입력 값을 주면, opaque code의 다양한 행동을 포함할 수 있도록 이로부터 다양한 입력 값을 생성한다. 그리고 입력 값에 대한 opaque code의 execution trace를 바탕으로 모델이 가지는 loop 구조를 유추한다. 이 때, execution trace는 opaque code를 실행했을 때 파라미터에 행해지는 메모리 연산을 저장하고 있다. 따라서 반복되는 메모리 연산를 찾아서 opaque code가 가질 수 있는 loop 구조를 유추한다. 그리고 입력 값의 종류를 나눠서 각 종류마다 유추한 loop 구조를 바탕으로 모델을 만든 후, 각 모델을 통합해 모든 입력 값을 고려한 하나의 모델을 만든다. 이 때, 같은 종류의 입력 값은 비슷한 execution trace를 가진다.

`Mimic`은 random search를 이용해 각 종류마다 모델을 자동으로 생성한다. 먼저 하나의 execution trace를 선택해 해당 trace와 유추한 loop 구조를 따르는 초기 모델을 만든다. 그리고 모델이 모든 execution trace를 따를 때까지 모델의 sub-expression을 임의로 바꾼다. 이 때, 주어진 execution trace와 모델의 execution trace를 비교하여 비슷할수록 낮은 fitness를 준다.

예를 들어, opaque code로 `Array.prototype.shift` 함수를 입력 값으로 `[‘a’, ‘b’, ‘c’, ‘d’]`를 사용해보자. `shift` 함수는 다음과 같이 행동한다:

```
var arr = ['a', 'b', 'c', 'd']
var x = arr.shift();
// x 변수는 'a' 값을 가지고 arr 변수는 ['b', 'c', 'd'] 값을 가진다.
```

이 때 execution trace와 해당 trace와 유추한 구조를 가지고 생성한 초기 모델은 그림 1과 같다.

```
read field 'length' of arg0 // 4
read field 0 of arg0 // 'a'
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

```
1  var n0 = arg0.length;
2  var n1 = arg0 [0];
3  for (var i = 0; i < 0; i += 1) {
4     var n2 = 1 in arg0;
5     if (true) {
6           var n3 = arg0[1];
7           arg0[0] = 'b';
8     } else {
9           delete arg0[2];
10    }
11 }
12 delete arg0[3];
13 arg0.length = 3;
14 return 'a';
```

그림 1. (좌) EXECUTION TRACE와 (우) 초기 모델의 예

초기 모델은 `arg0[0] = 'b'`와 같이 trace에 존재하는 실제 값을 사용했기 때문에, `shift` 함수를 제대로 표현하고 있지 않다. 따라서 모델 구조는 유지한 채 `arg[0]`이나 `'b'`와 같은 sub-expression을 임의의 expression으로 변경해 `shift` 함수를 표현하도록 만든다.

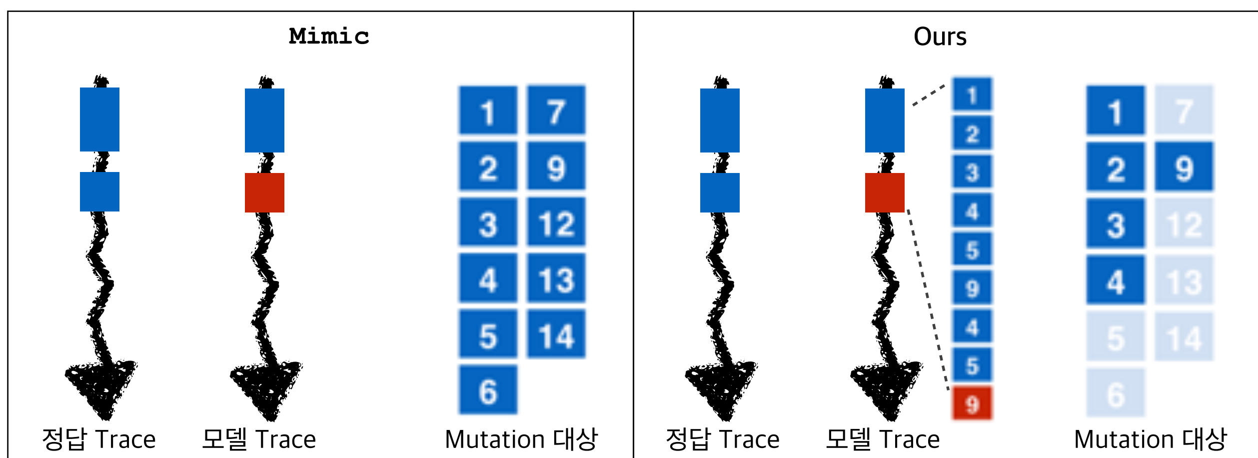

그림 2. (좌) MIMIC의 MUTATION 대상 (우) 이번 프로젝트에서 변경한 MUTATION 대상

3. Methodology

이번 그룹 프로젝트는 `Mimic`의 모델 탐색 전략을 수정하여 `Mimic` 보다 빠르게 모델을 생성하도록 만들었다. 특히, 변경하는 sub-expression을 고르는 방법, 즉, mutation 대상을 수정했다. 그림 2는 `Mimic`의 mutation 대상과 이번 그룹 프로젝트에서 변경한 mutation 대상을 비교하여 보여준다. 여기서 숫자는 그림 1의 초기 모델에서 라인 수를 의미한다. `Mimic`은 trace에 상관 없이 모든 expression이 mutation 대상으로, 이 중 임의로 하나를 골라 변경한다. 반면, 우리는 trace 결과에 따라 mutation 대상이 다르다. 정답 trace와 모델 trace가 처음 다른 메모리 연산을 할 때, 해당 연산과 해당 연산에 영향을 줄 수 있는 연산과 매칭되는 expression을 찾아, 이를 mutation 대상으로 사용한다. 영향을 줄 수 있는 연산은 적어도 실행한 연산이므로, 정답 연산과 다른 expression과 그 전에 실행한 expression이 mutation 대상이고, `Mimic`과 마찬가지로 이 중에서 임의로 하나를 선택해 변경한다. 이 때 틀린 expression 전에 실행된 expression 중에서 for, if, assign만 mutation 대상으로 고려했다.

`Mimic`은 여러 입력 값에 대해서 trace를 비교하기 때문에, 여러 개의 trace 결과를 mutation 대상 선정에 어떻게 반영할 지 결정해야 한다. 이번 프로젝트에선 세 가지 방법을 사용했다: (1) 가장 빨리 실패한 trace 결과 선택, (2) 가장 많이 실패한 trace 결과 선택, (3) 여러 개의 trace 결과를 누적하여 가중치 부과.

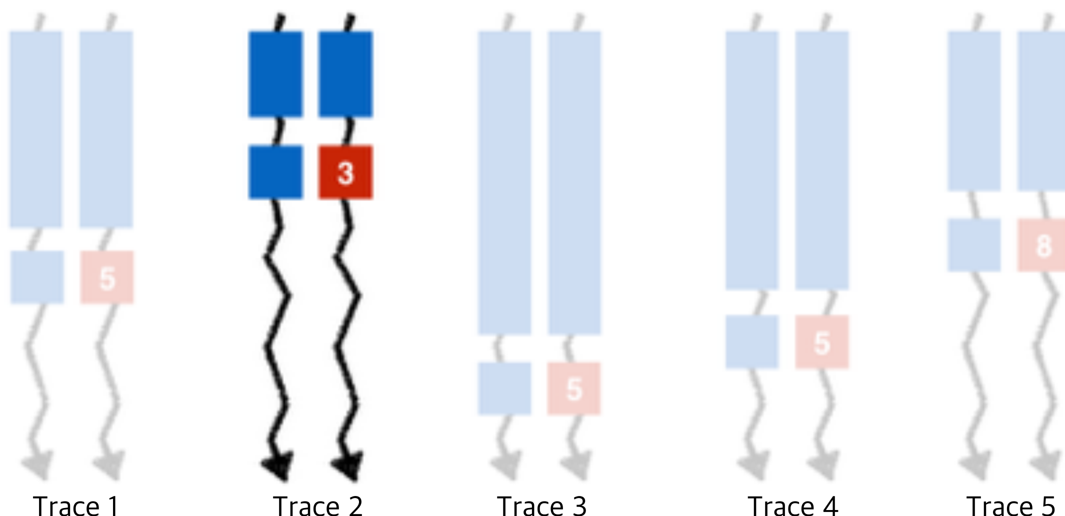

그림 3. 가장 빨리 실패한 결과 선택

3.1 가장 빨리 실패한 trace 결과 선택

여러 개의 trace 결과 중에서 가장 빨리 실패한 trace 결과에 따라 그림 2와 같이 mutation 대상을 선택한다. 정답 연산과 다른 모델 expression의 라인 수가 작을수록 빨리 실패했다고 한다. 따라서 그림 3에서 보여지듯, 모델 trace 결과마다 정답 trace와 처음으로 다른 메모리 연산을 한 expression을 찾아, expression의 라인 수가 가장 작은 trace 결과를 선택한다.

3.2. 가장 많이 실패한 trace 결과 선택

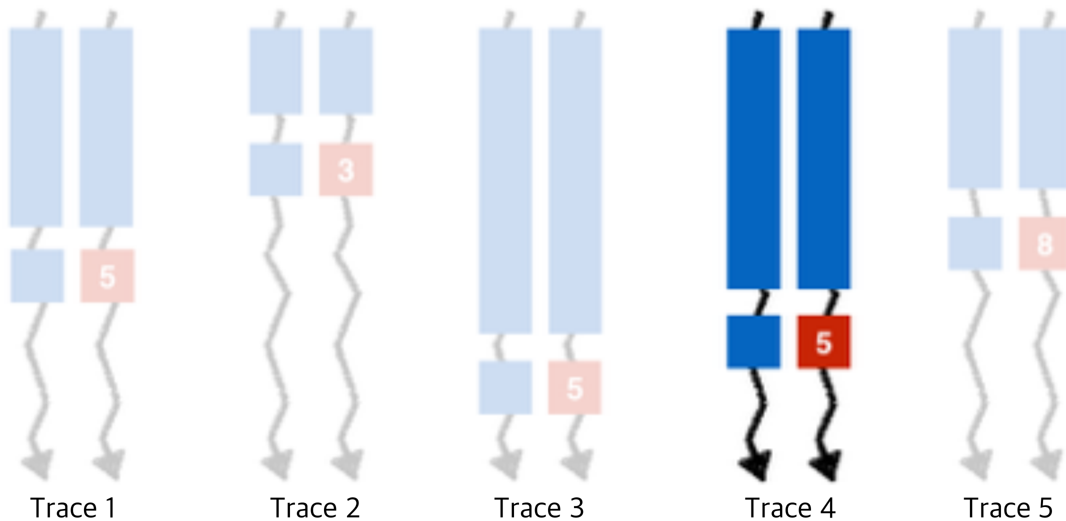

그림 4. 가장 많이 실패한 결과 선택

3.1의 방식과 거의 유사하나, 가장 빨리 실패한 trace 결과가 아닌 가장 많이 실패한 trace 결과에 따라 mutation 대상을 선택한다. 그림 4를 예로 들면, trace 1, trace 3, trace 4는 같은 곳에서 실패한다. 따라서 자주 실패한 trace 결과를 포함하고 있는 trace 1, trace 3, trace 4 중 임의로 하나의 결과 (trace 4)를 골라, 이를 바탕으로 mutation 대상을 선택한다.

3.3. 여러 개의 trace 결과를 누적하여 가중치 부과

3.1과 3.2는 여러 개의 trace 결과 중 하나를 골라 그림 2와 같은 방식으로 mutation 대상을 선택했다면, 3.3은 여러 개의 trace 결과를 누적하여 mutation 대상에 가중치를 줬다. 여러 trace 결과에 반복적으로 발생한 expression 일수록 mutation 될 가능성이 많도록 expression에 가중치를 부과했다. 그림 5를 예로 들면, 여러 trace에서 실행된 1 번째 expression은 mutation 될 확률이 15%인 반면 하나의 trace에서만 실행된 13 번째 expression은 mutation 될 확률이 3%이다.

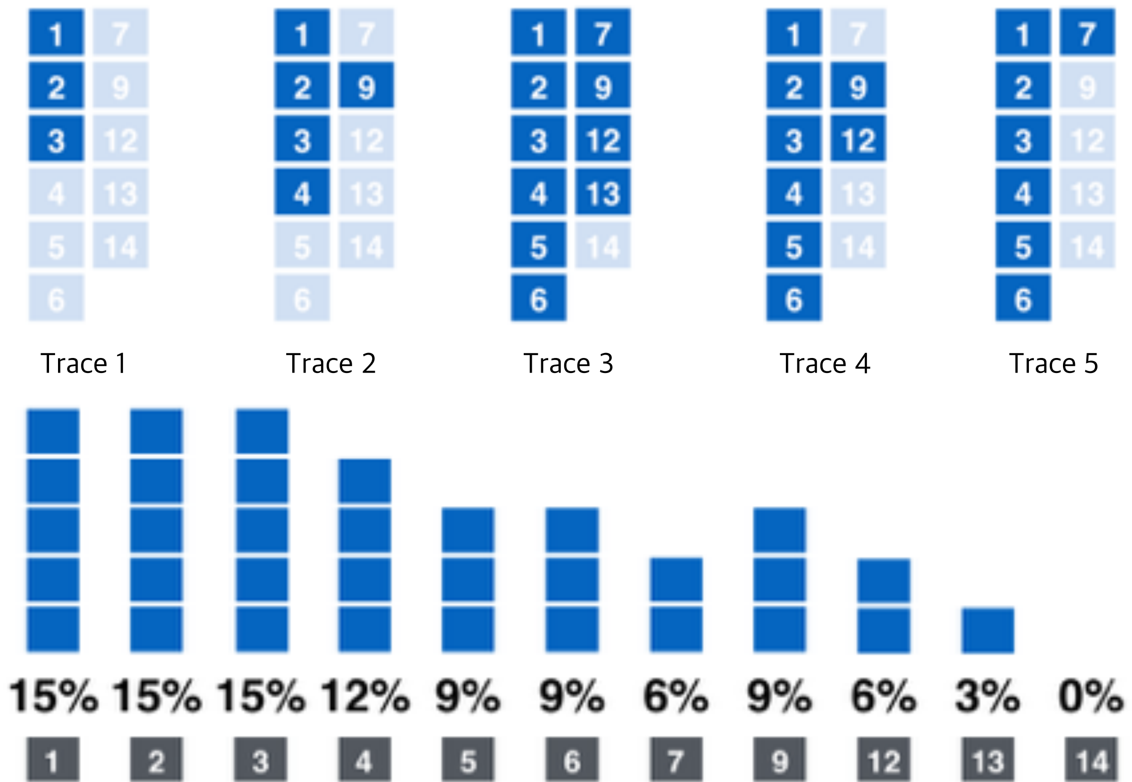

그림 5. 여러 개의 TRACE 결과를 누적하여 가중치 부과

4. Implementation

이번 프로젝트는 오픈 소스인 `Mimic`위에서 구현 되었다. Trace 결과를 mutation 대상 선택에 반영하기 위해 크게 세 가지 부분을 수정했다: (1) trace와 expression을 매칭하는 부분 (2) mutation 대상 부분 (3) 변경할 expression을 선택하는 부분.

먼저 trace에 존재하는 메모리 연산과 모델의 expression을 매칭할 수 있도록 생성한 모델을 instrument 했다. 생성한 모델은 statement 단위로 id를 가지고 (하위 expression은 같은 id를 가진다), expression이 실행될 때마다 id를 trace에 남기도록 만들었다. 따라서 trace가 기록한 메모리 연산과 매칭되는 expression을 찾으려면, 직전 기록에 남겨진 expression id를 살펴보면 된다.

위에서 설명한 매칭 방식을 각 trace 결과에 적용하여, 틀린 expression과 실행된 expression를 계산해 3장에서 설명한 방법들을 구현했다. 이 때 `Mimic`에선 mutation 대상이 동일한 가중치를 가지므로, 3.3장에서 설명한 가중치를 위해 변경할 expression을 선택하는 부분을 수정했다.

구현체와 실행 방법은 다음과 같다:

(1) 소스 코드
Mimic 위에서 구현된 이번 그룹 프로젝트의 소스 코드는 github repository에 올려놓았다:
https://github.com/jhnaldo/mimic

(2) 설치 방법
Mimic과 마찬가지로 도구를 실행하기 위해선 몇 가지 설치가 필요하다. 설치 방법은 github repository의 README를 따라하면 된다.

(3) 실행 방법
설치가 끝난 후, 다음과 같은 command line을 이용해 이번 그룹 프로젝트를 실행할 수 있다:
- 특정 opaque code에 (예. `shift`) 대한 모델 생성: scripts/example.py shift
- 실험에서 사용한 모든 자바스크립트 Array built-in 함수에 대한 모델 생성: scripts/experiment.py -n 1 (-n 이후 인자에 따라 실험 횟수를 변경할 수 있다.)

5. Evaluation

Mimic와 동일한 자바스크립트 Array built-in 함수에 대해서, Mimic과 이번 프로젝트에서 고안한 세 가지 방법을 비교해보았다. 실험은 Ubuntu Linux 16.04.1, Intel Core i7-6700K CPU 4 core, 32GB DDR4 사양에서 진행했고, Mimic과 세 가지 방법 모두 20번 실행하고 각 실행마다 모델 찾는 데 걸린 시간을 측정해 이를 평균냈다. 그림 6은 실험 결과를 보여준다. Mimic과 비교하여 걸린 시간이 10% 내외로 차이나면 노란색으로 표시하고, 10% 이상 빨라지면 초록색, 10% 이상으로 느려지면 빨간색으로 표시했다.

전체적인 실험 결과를 보면 이번 프로젝트에서 고안한 세 가지 방법 모두 Mimic과 마찬가지로 opaque code에 대한 자동 모델링을 생성할 수 있었고, 심지어 Mimic 보다 빠르게 모델을 생성했다. 세 가지 방법 중에서 (3.1) 가장 빨리 실패한 trace 결과를 이용한 방법이 전체적으로 봤을 때 가장 큰 성능 효과를 가져왔다. 이 방법은 `reduce`, `sum` 함수를 제외한 나머지 함수에 대해 Mimic 보다 빠르게 모델을 생성했다. 특히, `min` 함수에 대해선 약 4배 빨리 모델을 찾아냈다. (3.2) 가장 많이 실패한 trace 결과를 이용한 방법과 (3.3) 여러 trace 결과를 누적해 가중치를 부과한 방법은 전체적인 성능 향상은 3.1 방법 보다 작으나, 3.1 방법에서 성능 향상을 보지 못한 `push`, `reduce` 함수의 성능 향상을 보였다. 3.2 방법은 `push`, `sum` 함수를 제외한 나머지 함수에 대해 Mimic보다 빠르게 모델을 생성했

으며, 특히, 3.1과 3.3 방법에서 모두 성능 향상에 실패한 push에 대해 약 1.42배 성능 향상을 보였다.

3.3 방법은 전체적인 성능 향상은 다른 방법에 비해 작으나, 3.1과 3.2 방법에서 모두 성능 향상에 실패한 reduce에 대해 성능 향상을 보였고, min에 대해선 가장 빨리 모델을 생성해냈다.

Function	Mimic (sec)	Our 3.1 (sec)	Our 3.2 (sec)	Our 3.3 (sec)
every	156.3815	94.9445	102.799	117.0955
filter	63.6345	41.2025	46.946	61.7605
forEach	3.9285	2.4095	3.135	3.932
indexOf	137.5915	79.548	74.8195	136.6055
lastIndexOf	89.781	34.7835	72.53	44.079
map	11.3045	7.9345	7.508	9.4325
pop	2.211	1.8595	1.648	2.449
push	770.967	834.4475	543.055	847.99
reduce	27.4605	40.6815	33.4545	24.1775
reduceRight	321.6165	302.3315	274.1505	273.6115
shift	325.8775	136.5685	138.191	134.206
some	4.676	3.184	3.831	4.5485
max	242.5545	153.311	217.182	393.004
min	863.272	203.875	512.18	119.918
sum	59.6925	87.067	100.856	98.79
Total	3080.949	2024.148	2132.2855	2271.5995
Speed-up	1.0 X	1.52 X	1.44 X	1.36 X

그림 6. 실험 결과

6. Conclusion

이번 프로젝트에선 search-based approach를 이용해 정적 분석을 위한 opaque code 자동 모델링를 다뤘다. 특히, random search를 이용한 기존의 자동 모델링 도구인 Mimic 보다 빠르게 모델을 생성하는 데 초점을 맞췄다. 따라서 이번 프로젝트에선 모델과 정답의 execution trace를 비교한 결과를 이용해 다양한 방식으로 search space를 줄였고, 구현과 실험을 통해 줄인 search space을 가지고도 Mimic과 마찬가지로 opaque code에 대한 모델을 생성할 수 있음을 보였고, 심지어 Mimic 보다 평균적으로 약 1.44배 빠르게 모델을 생성함을 보였다.