질의-응답 프롬프트 기반 실용적인 테스트 오라클 생성

ANTLION: Practical Test Oracle Generation via Multi-turn LLM Compact Prompting

정지나, 김윤호

한양대학교 컴퓨터·소프트웨어학과(미래자동차-SW 융합전공), 컴퓨터·소프트웨어학과

Dept. of Computer and Software (Automotive–Computer Convergence), Dept. of Computer and Software, Hanyang University

snowgina00@hanyang.ac.kr, yunhokim@hanyang.ac.kr

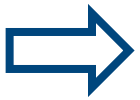
한양대학교

소프트웨어재난연구센터

Assert 작성의 어려움.

> 테스트 대상 메서드

```
public String normalizePath(String r) {
  boolean absol = r.startsWith("/") || r.startsWith("\\");
  String str = r.replace('\\', '/').replaceAll("/+", "/");
  String[] parts = str.split("/");
  Deque<String> s = new ArrayDeque<>();

  for (String p : parts) {
    if (p.isEmpty() || p.equals(".")) continue;

    if (p.equals("..")) {
      if (!s.isEmpty() && !s.peekLast().equals("..")) {
        s.removeLast();
      } else if (!absol) {
        s.addLast("..");
      }
    } else {
      s.addLast(p);
    }
  }

  String joined = String.join("/", s);
  if (absol) return "/" + joined;
  return joined.isEmpty() ? "." : joined;
}
```



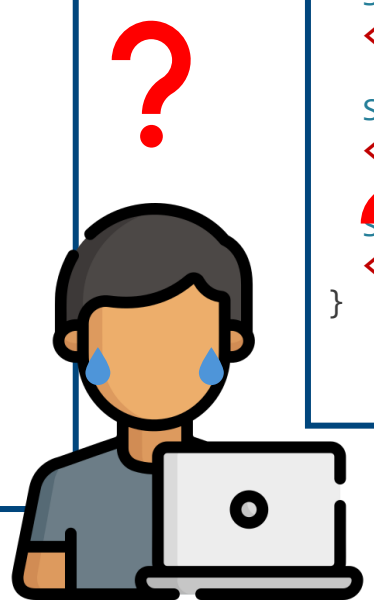
> 단위 테스트 코드

```
@Test
void normalizePath_Test() {
  PathUtil util = new PathUtil();

  String result1 = util.normalizePath("/../../a/..");
  <AssertPlaceHolder>

  String result2 = util.normalizePath("/a///b//./c//");
  <AssertPlaceHolder>

  String result3 = util.normalizePath("../a/b/../../c");
  <AssertPlaceHolder>

  String result4 = util.normalizePath("a/./b/..");
  <AssertPlaceHolder>

  String result5 = util.normalizePath("a/../../b");
  <AssertPlaceHolder>
}
```

Assert 작성의 어려움.

> 테스트 대상 메서드

```
public String normalizePath(String r) {
  boolean absol = r.startsWith("/") || r.startsWith("\\");
  String str = r.replace('\\', '/').replaceAll("/+", "/");
  String[] parts = str.split("/");
  Deque<String> s = new ArrayDeque<>();

  for (String p : parts) {
    if (p.isEmpty() || p.equals(".")) continue;

    if (p.equals("..")) {
      if (!s.isEmpty() && !s.peekLast().equals("..")) {
        s.removeLast();
      } else if (!absol) {
        s.addLast("..");
      }
    } else {
      s.addLast(p);
    }
  }

  String joined = String.join("/", s);
  if (absol) return "/" + joined;
  return joined.isEmpty() ? "." : joined;
}
```



> 단위 테스트 코드

```
@Test
void normalizePath_Test() {
  PathUtil util = new PathUtil();

  String result1 = util.normalizePath("/../../a/..");
  <AssertPlaceHolder>

  String result2 = util.normalizePath("/a///b//./c//");
  <AssertPlaceHolder>

  String result3 = util.normalizePath("../a/b/../../c");
  <AssertPlaceHolder>

  String result4 = util.normalizePath("a/./b/..");
  <AssertPlaceHolder>

  String result5 = util.normalizePath("a/../../b");
  <AssertPlaceHolder>
}
```

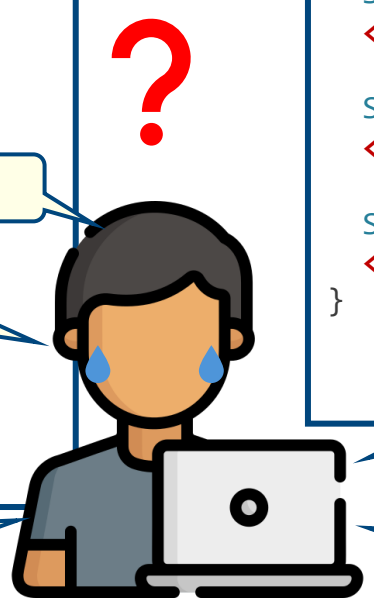
Assert 작성의 어려움.

> 테스트 대상 메서드

```java
public String normalizePath(String r) {
  boolean absol = r.startsWith("/") || r.startsWith("\\");
  String str = r.replace('\\', '/').replaceAll("/+", "/");
  String[] parts = str.split("/");
  Deque<String> s = new ArrayDeque<>();

  for (String p : parts) {
    if (p.isEmpty() || p.equals(".")) continue;

    if (p.equals("..")) {
      if (!s.isEmpty() && !s.peekLast().equals("..")) {
        s.removeLast();
      } else if (!absol) {
        s.addLast("..");
      }
    } else {
      s.addLast(p);
    }
  }

  String joined = String.join("/", s);
  if (absol) return "/" + joined;
  return joined.isEmpty() ? "." : joined;
}
```

> 단위 테스트 코드

```java
@Test
void normalizePath_Test() {
  PathUtil util = new PathUtil();

  String result1 = util.normalizePath("/../../a/..");
  <AssertPlaceHolder>

  String result2 = util.normalizePath("/a///b//./c//");
  <AssertPlaceHolder>

  String result3 = util.normalizePath("../a/b/../../c");
  <AssertPlaceHolder>

  String result4 = util.normalizePath("a/./b/..");
  <AssertPlaceHolder>

  String result5 = util.normalizePath("a/../../b");
  <AssertPlaceHolder>
}
```



```
assertEquals("/a/b", result2);
assertEquals("//", result1);
assertEquals("../d", result3);
assertEquals("/b", result6);
assertEquals("..", result4);
```

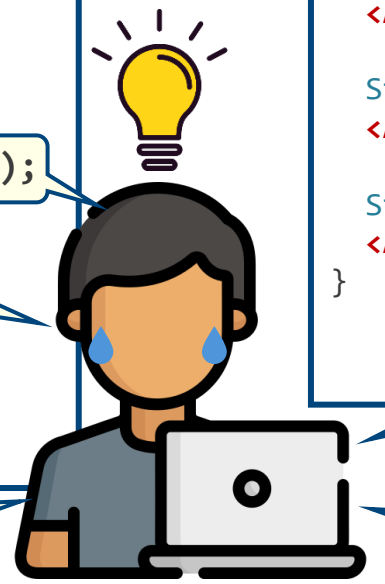
Assert 작성의 어려움.

> 테스트 대상 메서드

```java
public String normalizePath(String r) {
  boolean absol = r.startsWith("/") || r.startsWith("\\");
  String str = r.replace('\\', '/').replaceAll("/+", "/");
  String[] parts = str.split("/");
  Deque<String> s = new ArrayDeque<>();

  for (String p : parts) {
    if (p.isEmpty() || p.equals(".")) continue;

    if (p.equals("..")) {
      if (!s.isEmpty() && !s.peekLast().equals("..")) {
        s.removeLast();
      } else if (!absol) {
        s.addLast("..");
      }
    } else {
      s.addLast(p);
    }
  }

  String joined = String.join("/", s);
  if (absol) return "/" + joined;
  return joined.isEmpty() ? "." : joined;
}
```

> 단위 테스트 코드

```java
@Test
void normalizePath_Test() {
  PathUtil util = new PathUtil();

  String result1 = util.normalizePath("/../../a/..");
  <AssertPlaceHolder>

  String result2 = util.normalizePath("/a///b//./c//");
  <AssertPlaceHolder>

  String result3 = util.normalizePath("../a/b/../../c");
  <AssertPlaceHolder>

  String result4 = util.normalizePath("a/./b/..");
  <AssertPlaceHolder>

  String result5 = util.normalizePath("a/../../b");
  <AssertPlaceHolder>
}
```



assertEquals("/a/b/c", result2);

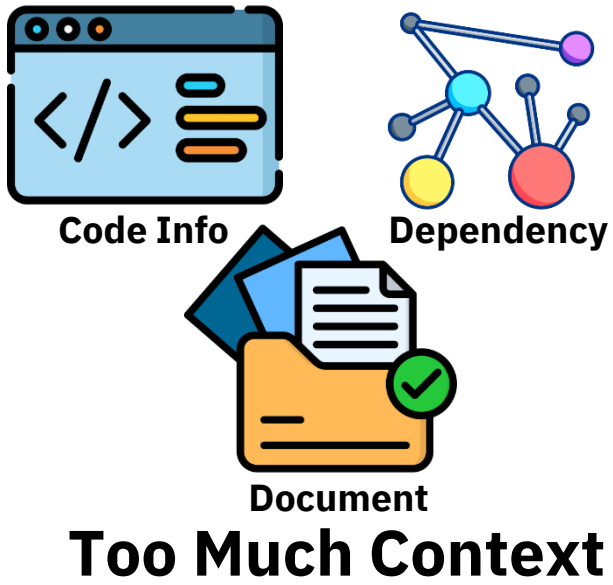
assertEquals("/", result1);

assertEquals("../c", result3);

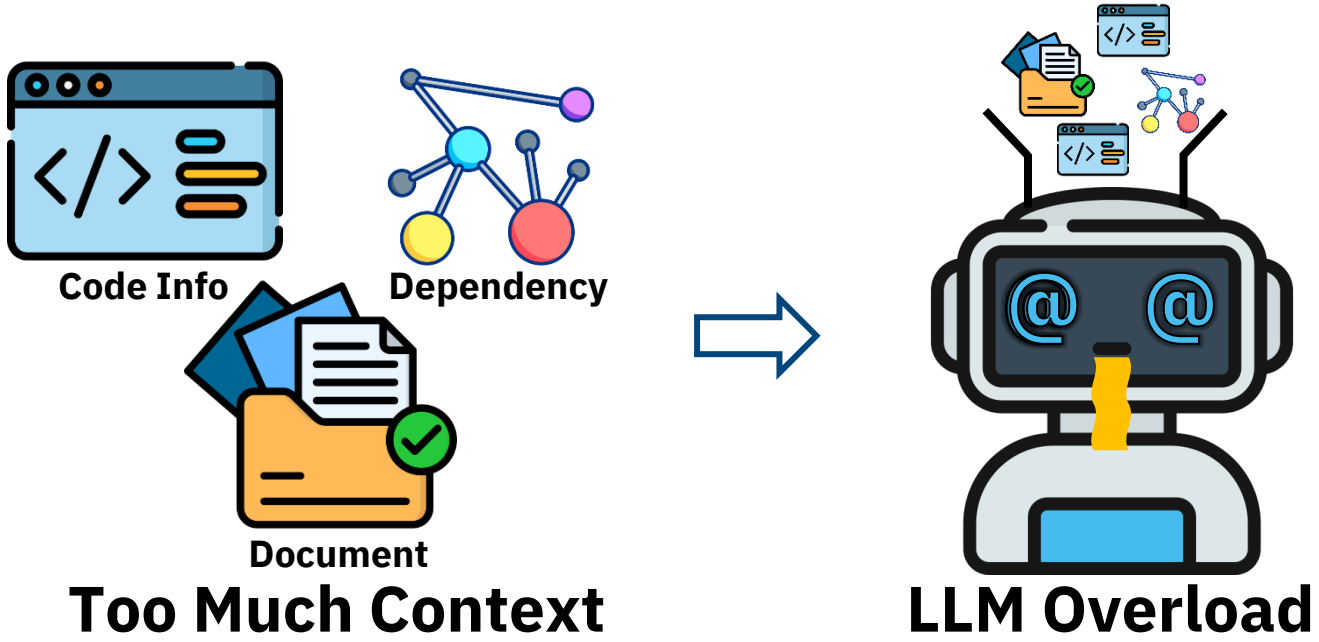
assertEquals("../b", result6);

assertEquals(".", result4);

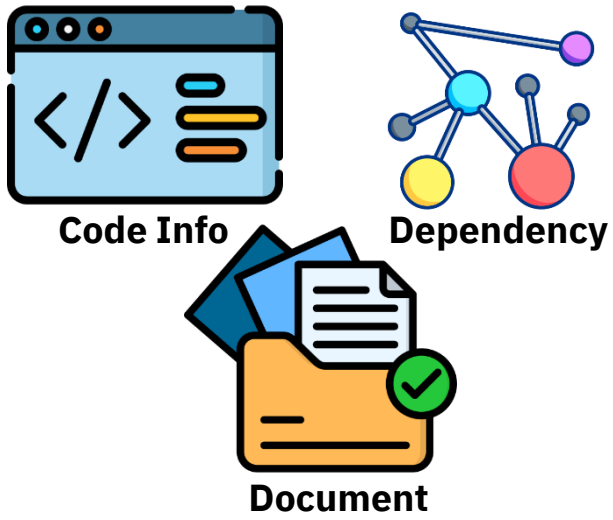
최신 LLM 기반 자동 테스트 오라클 생성 접근의 한계



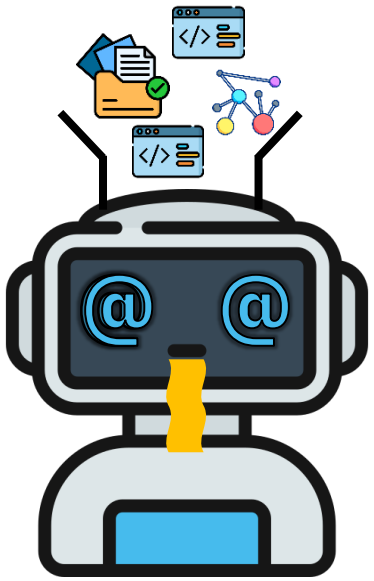
최신 LLM 기반 자동 테스트 오라클 생성 접근의 한계



최신 LLM 기반 자동 테스트 오라클 생성 접근의 한계



Too Much Context



LLM Overload

Ex1) 의미 없는 테스트(항상 통과)

```
Answer 1. assertTrue(true);
Answer 2. assertEquals(a, a);
Answer 3. assertNull(null);
```

Ex2) 기대 값을 지어냄(근거 없는 상수)

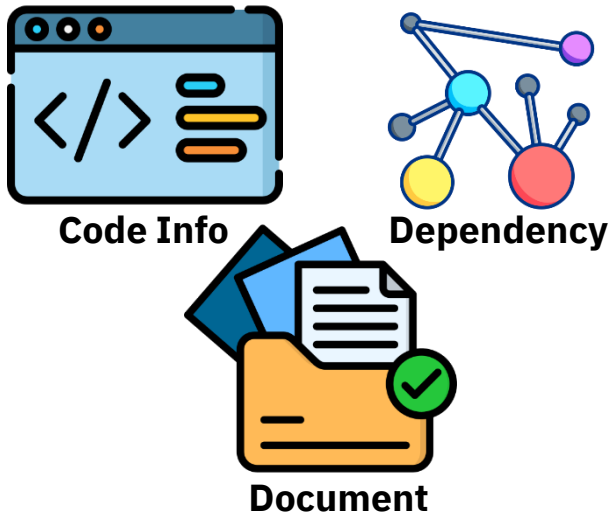
```
Answer 1. assertEquals(42, a);
Answer 2. assertEquals(7, a);
```

Ex3) 조건 반대

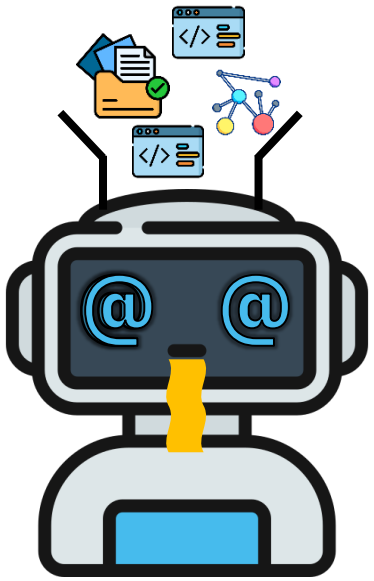
```
Answer 1. assertTrue(a > 0);
```

Hallucination

최신 LLM 기반 자동 테스트 오라클 생성 접근의 한계



Too Much Context



LLM Overload

Ex1) 의미 없는 테스트(항상 통과)

```
Answer 1. assertTrue(true);
Answer 2. assertEquals(a, a);
Answer 3. assertNull(null);
```

Ex2) 기대 값을 지어냄(근거 없는 상수)

```
Answer 1. assertEquals(42, a);
Answer 2. assertEquals(7, a);
```

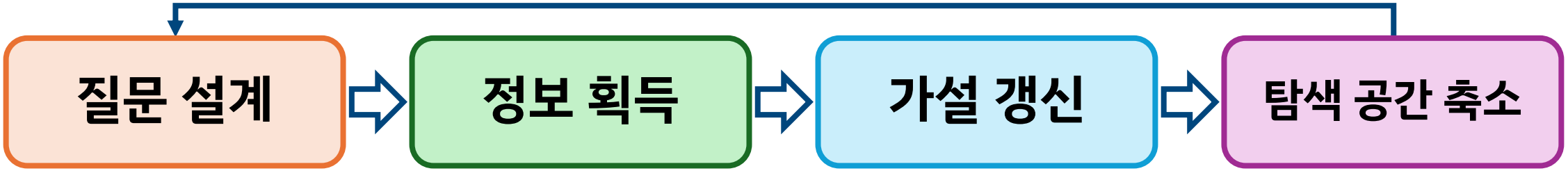
Ex3) 조건 반대

```
Answer 1. assertTrue(a > 0);
```

Hallucination

- 지나치게 긴 문맥은 LLM이 핵심 근거를 놓쳐, 그럴듯하지만 틀린 오라클(assert)을 생성.

Key Idea : 스무고개식 멀티 턴 질의-응답



- **최소한의 문맥을 유지하면서도, 체계적으로 불확실성을 해소하여, 일관된 추론을 유도.**
  - ✓ 노이즈를 방지 하기 위해 간결한 문맥 유지를 위해 멀티 턴 방식을 활용.
  - ✓ 위 과정을 통해서 단계적으로 정답에 다가가는 추론 과정이 포함된 프롬프트를 이용함.
  - ✓ 모든 정보를 고정적으로 제공하는 대신, 불확실성을 해소 하기위한 단계적 정보 수집 과정.

ANTLION: 멀티 턴 질의-응답 프롬프트 기반 테스트 오라클 생성

입력 컨텍스트

- **테스트 대상 메서드(MUT)**

```
public int add(int a, int b) {
  return a + b;
}
```

- **테스트 접두 코드(Prefix)**

```
@Test
void addTest() {
  Calculator c = new Calculator();
  int result = c.add(3, 4);
  <AssertPlaceHolder>
}
```

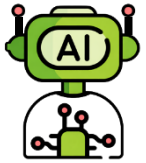
- **질문 목록**

✓ **질문 A.** 테스트 **대상 메서드의 의도** 설명
✓ **질문 B.** 테스트 **접두 코드의 의도** 설명
✓ **질문 C.** 테스트 대상 메서드의 **CFG** 제시
✓ **질문 D.** 오라클의 **후보(top-k)** 제시
✓ **질문 E.** 테스트 대상 메서드의 **Javadoc** 제시

사용자 User

문제.

정확한 오라클 구문으로 대체되어야 할 **“<AssertPlaceHolder>”**가 포함된 테스트 접두 코드와 테스트 대상 메서드를 준비했습니다.
당신의 임무는 그 빈칸을 채울 가장 적절한 **JUnit 4 단언문을 추론**하는 것입니다.
선택적으로 질문 목록을 사용해 추가 정보를 질문할 수 있고, 확신이 있다면 곧바로 정답을 제출해도 됩니다.
지금부터 **Junit 4 형식**의 **테스트 오라클 추론**을 시작해주세요.

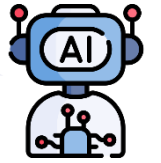


답변 AI

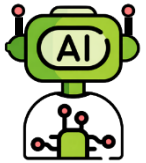
질문 1.

A. 테스트 대상 메서드(MUT)의 의도를 설명해주세요.

답변. 두 정수를 입력으로 받아 그 합을 반환 하는 순수 덧셈 함수 입니다.



질문 AI

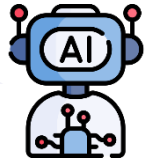


답변 AI

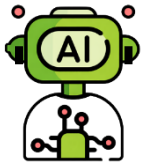
질문2.

B. 테스트 접두 코드(Prefix)의 의도를 설명해주세요.

답변. 테스트 대상 메서드(MUT)가 두 개의 정수 인수 3와 4로 호출되는 흐름으로, 접두사의 목적은 메서드가 이러한 입력의 합을 올바르게 계산하는지 확인하는 것입니다.



질문 AI

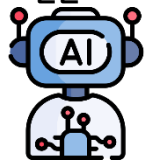


답변 AI

질문 3.

D. 테스트 오라클의 후보를 제시해주세요.

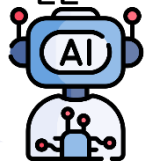
답변. assertEquals(7, result);
assertEquals(5, result);
assertEquals(6, result);



질문 AI

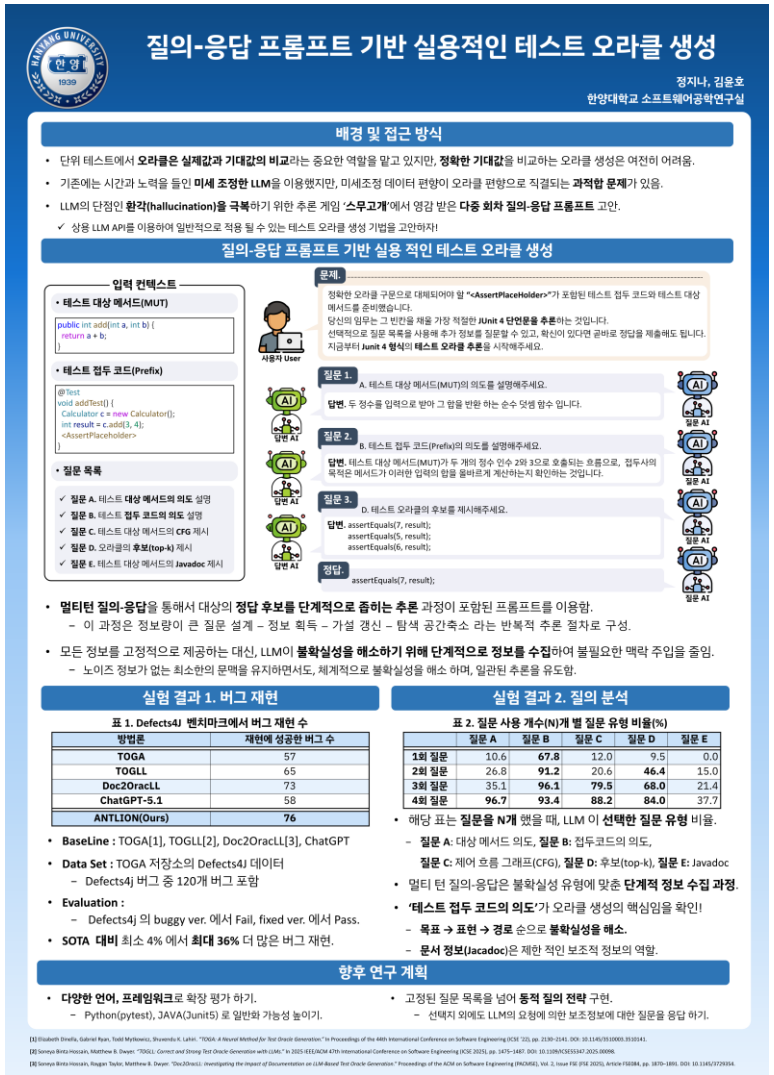
정답.

assertEquals(7, result);



질문 AI

포스터 많은 관심 바랍니다.



질의-응답 프롬프트 기반 실용적인 테스트 오라클 생성

정지나, 김윤호
한양대학교 소프트웨어공학연구실

배경 및 접근 방식

- 단위 테스트에서 **오라클은 실제값과 기대값의 비교**라는 중요한 역할을 맡고 있지만, **정확한 기대값**을 비교하는 오라클 생성은 여전히 어려움.
- 기존에는 시간과 노력을 들인 **미세 조정한 LLM**을 이용했지만, 미세조정 데이터 편향이 오라클 편향으로 직결되는 **과적합 문제**가 있음.
- LLM의 단점인 **환각(hallucination)을 극복**하기 위한 추론 게임 **'스무고개'**에서 영감 받은 **다중 회차 질의-응답 프롬프트** 고안.
  - ✓ 상용 LLM API를 이용하여 일반적으로 적용 될 수 있는 테스트 오라클 생성 기법을 고안하자!

질의-응답 프롬프트 기반 실용 적인 테스트 오라클 생성

입력 컨텍스트
- 테스트 대상 메서드(MUT)
- 테스트 접두 코드(Prefix)
- 질문 목록
  - ✓ 질문 A. 테스트 **대상 메서드의 의도** 설명
  - ✓ 질문 B. 테스트 **접두 코드의 의도** 설명
  - ✓ 질문 C. 테스트 대상 메서드의 **CFG** 제시
  - ✓ 질문 D. 오라클의 **후보(top-k)** 제시
  - ✓ 질문 E. 테스트 대상 메서드의 **Javadoc** 제시

문제.
정확한 오라클 구문으로 대체되어야 할 **"<AssertPlaceHolder>"**가 포함된 테스트 접두 코드와 테스트 대상 메서드를 준비했습니다.
당신의 임무는 그 빈칸을 채울 가장 적절한 **JUnit 4 단언문을 추론**하는 것입니다.
선택적으로 질문 목록을 사용해 추가 정보를 질문할 수 있고, 확신이 있다면 곧바로 정답을 제출해도 됩니다.
지금부터 **Junit 4 형식**의 **테스트 오라클 추론**을 시작해주세요.

질문 1. A. 테스트 대상 메서드(MUT)의 의도를 설명해주세요.
답변. 두 정수를 입력으로 받아 그 합을 반환 하는 순수 덧셈 함수 입니다.

질문 2. B. 테스트 접두 코드(Prefix)의 의도를 설명해주세요.
답변. 테스트 대상 메서드(MUT)가 두 개의 정수 인수 2와 3으로 호출되는 흐름으로, 접두사의 목적은 메서드가 이러한 입력의 합을 올바르게 계산하는지 확인하는 것입니다.

질문 3. D. 테스트 오라클의 후보를 제시해주세요.
답변. assertEquals(7, result);
assertEquals(5, result);
assertEquals(6, result);

정답. assertEquals(7, result);

- **멀티턴 질의-응답**을 통해서 대상의 **정답 후보를 단계적으로 좁히는 추론** 과정이 포함된 프롬프트를 이용함.
  - 이 과정은 정보량이 큰 질문 설계 – 정보 획득 – 가설 갱신 – 탐색 공간축소 라는 반복적 추론 절차로 구성.
- 모든 정보를 고정적으로 제공하는 대신, LLM이 **불확실성을 해소하기 위해 단계적으로 정보를 수집**하여 불필요한 맥락 주입을 줄임.
  - 노이즈 정보가 없는 최소한의 문맥을 유지하면서도, 체계적으로 불확실성을 해소 하며, 일관된 추론을 유도함.

실험 결과 1. 버그 재현

표 1. Defects4J 벤치마크에서 버그 재현 수

방법론	재현에 성공한 버그 수
TOGA	57
TOGLL	65
Doc2OracLL	73
ChatGPT-5.1	58
ANTLION(Ours)	76

- **BaseLine :** TOGA[1], TOGLL[2], Doc2OracLL[3], ChatGPT
- **Data Set :** TOGA 저장소의 Defects4J 데이터
  - Defects4j 버그 중 120개 버그 포함
- **Evaluation :**
  - Defects4j 의 buggy ver. 에서 Fail, fixed ver. 에서 Pass.
- **SOTA 대비** 최소 4% 에서 **최대 36%** 더 많은 버그 재현.

실험 결과 2. 질의 분석

표 2. 질문 사용 개수(N)개 별 질문 유형 비율(%)

	질문 A	질문 B	질문 C	질문 D	질문 E
1회 질문	10.6	**67.8**	12.0	9.5	0.0
2회 질문	26.8	**91.2**	20.6	**46.4**	15.0
3회 질문	35.1	**96.1**	**79.5**	**68.0**	21.4
4회 질문	**96.7**	**93.4**	**88.2**	**84.0**	37.7

- 해당 표는 **질문을 N개** 했을 때, LLM 이 **선택한 질문 유형** 비율.
  - **질문 A**: 대상 메서드 의도, **질문 B:** 접두코드의 의도,
    질문 C: 제어 흐름 그래프(CFG), **질문 D:** 후보(top-k), **질문 E:** Javadoc
- 멀티 턴 질의-응답은 불확실성 유형에 맞춘 **단계적 정보 수집 과정**.
- **'테스트 접두 코드의 의도'**가 오라클 생성의 핵심임을 확인!
  - **목표 → 표현 → 경로** 순으로 **불확실성을 해소.**
  - **문서 정보(Jacadoc)**은 제한 적인 보조적 정보의 역할.

향후 연구 계획

- **다양한 언어, 프레임워크**로 확장 평가 하기.
  - Python(pytest), JAVA(Junit5) 로 일반화 가능성 높이기.
- 고정된 질문 목록을 넘어 **동적 질의 전략** 구현.
  - 선택지 외에도 LLM의 요청에 의한 보조정보에 대한 질문을 응답 하기.

[1] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, Shuvendu K. Lahiri. "TOGA: A Neural Method for Test Oracle Generation." In Proceedings of the 44th International Conference on Software Engineering (ICSE '22), pp. 2130–2141. DOI: 10.1145/3510003.3510141.
[2] Soneya Binta Hossain, Matthew B. Dwyer. "TOGLL: Correct and Strong Test Oracle Generation with LLMs." In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025), pp. 1475–1487. DOI: 10.1109/ICSE55347.2025.00098.
[3] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, Omer Tripp. "A Deep Dive into Large Language Models for Automated Bug Localization and Repair." Proceedings of the ACM on Software Engineering (FSE '24), Vol. 1, Article 66, pp. 1–23. DOI: 10.1145/3660773.