Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests

Amirhossein Deljouyi and Roham Koohestani and Maliheh Izadi and Andy Zaidman

icse2025

概要

タイトル

- 大規模言語モデルを活用した生成されたユニットテストの理解性向上に関する研究

現状と課題

- ユニットテストは，ソフトウェアの頻出を高めるうえで大きな役割を果たすが，多くの労力がかかる.
- 様々なシステムを用いてテストを自動生成する方法が研究されてきた.
- 高精度のテストは生成できるが，可読性がかなり低い.

概要

```
@Test
public void
↪  testCreatesWeaponGameDataTaking6ArgumentsAndCallsEquals3() {
  WeaponGameData jSWeaponData_WeaponGameData0 = new
    WeaponGameData(35, 17, 35, "N&zMn$@6gffi<");
  WeaponGameData jSWeaponData_WeaponGameData1 = new
    WeaponGameData(35, 35, 35, "N&zMn$@6gffi<");
  boolean boolean0 = jSWeaponData_WeaponGameData0.
    equals(jSWeaponData_WeaponGameData1);
  assertFalse(boolean0);
}
```

Fig. 1. Motivating Example

1. 意味を持たないテストデータ
2. 不明確な変数名
3. 理解を助けるコメントの欠如
4. 目的を反映していないテスト名

アプローチ

UTGen

- 既存のテスト生成フレームワークとLLMを統合することで理解しやすいユニットテストを生成する.

1. 理解しやすいテストデータの使用
2. テストの内容に沿った変数名の使用
3. 目的を反映したテスト名の採用

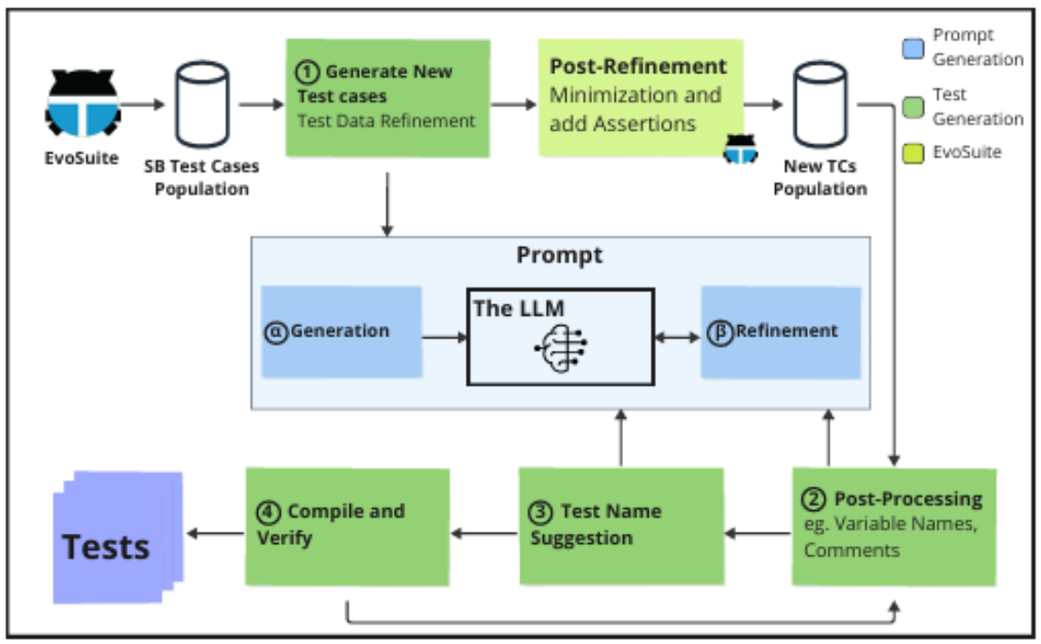

Fig. 2. Overview of the UTGen approach

UTGen

プロンプト

1. ペルソナを付与
2. タスクを指示
3. 構造化したステップ
4. 入出力の指定

```
① <<SYS>>
     You are a (ADJECTIVE) developer focusing on (TASK AT HAND)
  <</SYS>>
  [INST]
② Your task is to (TASK)
  To achieve this, you should follow these structured steps:
③   ** Detailed steps for analysis or improvement, emphasizing **
    1. Careful reading and understanding of the provided code.
    2. Identification of key functionalities or aspects requiring
    ↪  attention.
    3. Formulation or modification of specific elements (e.g., test
    ↪  names, data, comments) to enhance clarity, descriptiveness,
    ↪  or functionality.
    4. Adherence to coding standards and best practices, such as
    naming conventions or comment clarity.
④ The code section requiring your attention is delineated by [CODE]
  ↪  and [/CODE] tags.
  Your response, (whether it be a name, modified code, or
  comments), should be placed between the (OPENING TAG) and
  ↪  (CLOSING TAG).
  [CODE]
    The code segment to be analyzed or improved, dynamically
    ↪  inserted during execution
  [/CODE]
```

Fig. 3. Prompt template for UTGen

UTGen

テストデータの洗練

- テスト生成フレームワークでテストを生成した後，使用されるデータをより人間が理解しやすいデータに変更する.
- 例：「N&zMn$@6gffi<」　→　「Ninja Sword」

① Test Data Refinement

```
WeaponGameData weaponGameData0 =
new WeaponGameData(35, 35, 35,
↪  "N&zMn$@6gffi<");
```

```
WeaponGameData weaponGameData0 =
new WeaponGameData(35, 35, 35,
↪  "Ninja Sword");
```

UTGen

後処理

- Given-When-Then規約を使用してコメントを追加する.
- 理解しやすいように変数名を変更する.

※データそのものや，ロジックは変更しない.

- 例：「weaponGameData0」　→　「defaultWeapon」
- 例：「weaponGameData1」　→　「customWeapon」

② Post Processing; Variables + Comments

```
// Given: We have two instances of JSWeaponData.WeaponGameData with
↪  different values for minDmg
WeaponGameData defaultWeapon = new WeaponGameData(35, 35, 35, "Ninja Sword");
WeaponGameData customWeapon = new WeaponGameData(35, 17, 35, "Ninja Sword");

// When: We call equals() method on the two instances with the default
↪  WeaponGameData instance
boolean equals = defaultWeapon.equals(customWeapon);

// Then: The result of equals is false, since the two instances have
↪  different values for minDmg
assertFalse(equals, "The two instances should not be equal");
```

UTGen

テストメソッド名の提案

- テストの機能を反映した説明的な名前を生成する.
- すでに似た名前のものがあれば，別のものを生成する.
- 例：「testCreatesWeaponGameDataTaking6ArgumentsAndCallsEquals3」
  →　「testEqualsWithDifferentMinDmgValues」

③ Suggest a Test Method Name Based on the Test Body

```
public void testEqualsWithDifferentMinDmgValues() { ... }
```

UTGen

```
@Test
public void
↪  testCreatesWeaponGameDataTaking6ArgumentsAndCallsEquals3() {
  WeaponGameData jSWeaponData_WeaponGameData0 = new
    WeaponGameData(35, 17, 35, "N&zMn$@6gffi<");
  WeaponGameData jSWeaponData_WeaponGameData1 = new
    WeaponGameData(35, 35, 35, "N&zMn$@6gffi<");
  boolean boolean0 = jSWeaponData_WeaponGameData0.
    equals(jSWeaponData_WeaponGameData1);
  assertFalse(boolean0);
}
```

Fig. 1. Motivating Example

```
@Test
public void testEqualsWithDifferentMinDmgValues() {
  // Given: We have two instances of WeaponGameData with
  ↪  different values for minDmg
  WeaponGameData defaultWeapon =
    new WeaponGameData(35, 35, 35, "Ninja Sword");
  WeaponGameData customWeapon =
    new WeaponGameData(35, 17, 35, "Ninja Sword");

  // When: We call equals() method on the two instances
  ↪  with the default WeaponGameData instance
  boolean equals = defaultWeapon.equals(customWeapon);

  // Then: The result of equals is false, since the two
  ↪  instances have different values for minDmg
  assertFalse(equals,
  ↪  "The two instances should not be equal");
}
```

効果検証

RQ1　効果的なUTを生成できるか

- コンパイル率，カバレッジ，合格率の三つの観点において，EvoSuiteと比較.

結果

- コンパイル率：改善に成功(73%)，コンパイル不可(12%)，エンハンスメント停滞(16%)
- カバレッジ：命令24.43%(-0.6)，ブランチ17.87%(-0.81)
- 合格率：73.27%(-5.74%)

効果検証

RQ2　バグ修正への影響

- 32名のプログラマで，バグ修正における影響があるかを調査.
- 2つのJavaクラスを用意し，クロスオーバーデザインを使用.

結果

- 成功率
  - 統計的にも差があり，コーエンのdも0.59と，中程度の効果がある.
  - 最大約30%の増加.
- 時間効率
  - 統計的な差だけでなく，d=0.92と大きな効果があることがわかる.
  - 最大約20%の増加

効果検証

RQ3　理解へ及ぼす影響

- プログラマに事後アンケートを行い，どの要素がテストの理解に影響を及ぼすのかを調査.

- 観点１　理解度がバグ修正に与える影響
  - 78.1%がテストの理解度がバグ修正に重要だと答え，明確なテストスイートが最も重要だと答えた.
- 観点２　理解度に影響する要素
  - どの要素も重要であるが，テスト名が最も重要であるという回答が最も多かった.
- 観点３　テストケースの品質
  - 高いと答えた回答者が65.6%(+15.6%)であり，変数名の変更が最も評価が高かった.

課題

1. エンハンスメント停滞
   - LLMがテストケースを改善できない例があった.
   - 多数のパラメータを持つメソッドや密接に結合したメソッドで特に顕著であった.

2. LLMの最適化
   - 汎用的なLLMではなく，ファインチューニングしたものを使用することが求められる.

3. カバレッジの維持
   - 大きな差はなかったものの，カバレッジの低下がみられた.

結論

UTGen

- 既存のテスト生成フレームワークとLLMを統合することで，UTの理解度を向上させるアプローチ.

結果

- バグ修正においてプログラマの理解を助け，修正効率を向上させた.
- テストにおいてはカバレッジが重要視されてきたが，テストの理解度もテストの品質に含めても良いほど重要な要素であるといえる.