Praca naukowa dotycząca porównania istniejących analizatorów statycznych

Maryia Babinskaya, Alicja Bieżychudek, Mariia Saltykova
Vladyslav Khabanets, Tomasz Kosmulski, Maksim Zdobnikau
Uniwersytet Jagielloński

29 marca 2025

Streszczenie

Tło: Analizatory statyczne odgrywają kluczową rolę w zapewnieniu jakości oprogramowania, wykrywając potencjalne błędy i luki w kodzie źródłowym. Pomimo ich szerokiego zastosowania, istnieje potrzeba systematycznego porównania ich skuteczności, wydajności i funkcjonalności.
Cel: Celem niniejszej pracy jest porównanie istniejących analizatorów statycznych pod kątem ich możliwości, ograniczeń i zastosowań w różnych kontekstach programistycznych.
Metody:
Wyniki:
Wnioski:
Słowa kluczowe: analizatory statyczne, jakość oprogramowania

1 Wprowadzenie

Analizatory statyczne są nieodzownym narzędziem w procesie tworzenia oprogramowania, umożliwiając wykrywanie błędów na wczesnych etapach rozwoju. Pomimo ich szerokiego zastosowania, istnieje wiele wyzwań związanych z ich skutecznością i wydajnością. Niniejsza praca ma na celu porównanie istniejących analizatorów statycznych, aby pomóc deweloperom w wyborze najbardziej odpowiedniego narzędzia dla ich potrzeb.

2 Powiązane prace

3 Metodologia

Badanie systematycznie porównywało narzędzia statycznej analizy kodu poprzez:

1. **Projekt bazy danych**: Stworzenie relacyjnego schematu (`metryki`) do przechowywania metryk, narzędzi, języków programowania i ich powiązań. Schemat obejmował 15 tabel znormalizowanych do trzeciej postaci normalnej (3NF), z uwzględnieniem relacji wiele-do-wielu między narzędziami a metrykami.

2. **Gromadzenie danych**: Ekstrakcja metryk (złożoność cyklomatyczna, duplikacje, pokrycie kodu itp.) z 15 narzędzi analitycznych, w tym SonarQube, Qodana, PMD i Lizard. Dane zbierano dla czterech języków programowania (C++, Java, Python, TypeScript) na dwóch poziomach szczegółowości: całych repozytoriów oraz pojedynczych plików źródłowych.

3. **Normalizacja**: Strukturyzacja danych w tabelach (`tools`, `metricNames`, `metricValues`) z zastosowaniem technik transformacji danych, w tym:

   - Standaryzacja nazw metryk między różnymi narzędziami
   - Konwersja jednostek miar do wspólnego formatu
   - Mapowanie podobnych metryk z różnych analizatorów

4. **Automatyzacja**: Opracowanie procedury `InsertMetricValue2` w celu standaryzacji wstawiania metryk. Procedura realizuje:

   - Automatyczne wyszukiwanie ID języka, narzędzia i metryki
   - Walidację spójności danych przed wstawieniem
   - Rejestrację źródła danych (`sourceID`)
   - Obsługę relacji między encjami

4 Przegląd analizatorów

W badaniu uwzględniono następujące narzędzia analityczne:

- **Narzędzia wielojęzykowe**:
  - Qodana (wsparcie dla 60+ języków)
  - SonarQube (analiza jakości kodu i wykrywanie podatności)

  - PVS-Studio (C/C++, C#, analiza zgodności z MISRA)

- **Narzędzia specjalistyczne**:
  - **Java**: PMD, Checkstyle, JaCoCo (pokrycie kodu)
  - **Python**: PyDev, PyCharm, Pylint
  - **C/C++**: cppdepend, OCLint, Lizard
  - **TypeScript**: FTA, cyclomatic-complexity
  - **Analiza repozytoriów**: FREGE (analiza metadanych projektów open-source)

Kluczowe obserwacje

- **Różnorodność metryk**: Wykryto 7 różnych implementacji złożoności cyklomatycznej i 4 różne podejścia do wykrywania duplikatów kodu. Na przykład:
  - Lizard wymaga minimum 100 tokenów dla uznania fragmentu za duplikat
  - SonarQube stosuje próg 10 linii kodu
- **Specjalizacja**: 60% narzędzi skupia się na konkretnych językach, np.:
  - JaCoCo dedykowane wyłącznie dla Javy
  - PyLint specjalizowany w analizie kodu Python
- **Integracje**: 30% analizatorów to samodzielne narzędzia (Lizard, PMD), podczas gdy 20% jest zintegrowanych z IDE (PyCharm, PyDev). Pozostałe 50% działa jako niezależne platformy analityczne (SonarQube, Qodana).

5 Analiza danych

Kluczowe wnioski z analizy 47 rekordów z tabeli `metricDiscrepancies`:

1. **Różnice w pomiarach**:
   - Średnia różnica dla metryk złożoności wyniosła %
   - Maksymalna rozbieżność w wykrywaniu duplikatów osiągnęła % (Lizard vs SonarQube)
   - Dla metryk pokrycia kodu odnotowano różnice do %
2. **Różnice definicyjne**:
   - JaCoCo wyklucza gałęzie wyjątków z pokrycia kodu, podczas gdy SonarQube je uwzględnia

- PMD i Checkstyle stosują różne algorytmy obliczania złożoności NPATH
- 5 narzędzi stosuje różne progi dla wykrywania długich metod

3. **Stronniczość narzędzi**:
   - CppDepend konsekwentnie zaniża liczbę linii kodu o 15-20% w porównaniu do OCLint
   - Analizatory zintegrowane z IDE (PyCharm) wykazują tendencję do zawyżania metryk jakości kodu
   - Narzędzia oparte na analizie statycznej (PVS-Studio) dają bardziej konserwatywne wyniki

6 Wnioski

1. **Konieczność standaryzacji**:
   - Należy opracować wspólny słownik metryk uwzględniający różnice definicyjne
   - Wymagana jest unifikacja metod obliczeniowych dla kluczowych metryk
2. **Wybór narzędzi**:
   - Dla projektów wielojęzykowych rekomenduje się kombinację SonarQube + Qodana
   - W projektach Java warto stosować JaCoCo do pokrycia kodu i PMD do analizy jakości
   - Dla C++ najbardziej spójne wyniki daje połączenie PVS-Studio i cppdepend
3. **Walidacja i dalsze badania**:
   - Niestandardowe skrypty (np. dla OCLint) wymagają standaryzacji metod pomiaru
   - Konieczne jest rozszerzenie badań o analizę false-positive/false-negative
   - Warto zbadać wpływ wersji narzędzi na stabilność wyników

7 Dyskusja

Wyniki sugerują, że wybór analizatora statycznego powinien być uzależniony od konkretnych potrzeb projektowych.

Podziękowania

Autorzy pragną podziękować Uniwersytetowi Jagiellońskiemu za wsparcie w realizacji niniejszego projektu.

Oświadczenie o wkładzie autorów

Maryia Babinskaya:
Alicja Bieżychudek:
Mariia Saltykova:
Vladyslav Khabanets:
Tomasz Kosmulski:
Maksim Zdobnikau:

Oświadczenie o konflikcie interesów

Autorzy deklarują brak konfliktu interesów.

Dostępność danych

Dane użyte w badaniu są dostępne na platformie example.com.

Literatura