Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

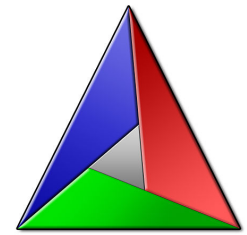
Werkzeuggestütztes Testen

Phillip Raffeck, Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

Wintersemester 2020

Testfallintegration mit CMake



- Integration von Tests im Softwareprojekt
- Automatisierte Ausführung und Auswertung von Testläufen
- Konfigurationsdatei: `tests/CMakeLists.txt`
  - Ausführbares Target:
    `add_executable(plus_test plus_test.c)`
  - Hinzubinden der zu testenden Bibliothek:
    `target_link_libraries(plus_test mathe)`
  - Bekanntmachen als Testfall:
    `add_test(MatheTest_PLUS plus_test)`
- Ausführung der Tests: `make && make test`
- Automatische Testauswertung:
  - Anhand Rückgabewert (0 → OK, -1 → Fehler)
  - Notfalls auch Parsen von Ausgaben
- Ausgaben der Tests (`(f)printf`) protokolliert in Datei
  `Testing/Temporary/LastTest.log`

Testfallintegration mit CMake

- Tests sind Programme im Unterverzeichnis `tests`

```
tests
|-- CMakeLists.txt
|-- priority_queue_test1.c
|-- priority_queue_test2.c
`-- priority_queue_test_malloc.c
```

- Die Datei `tests/CMakeLists.txt` definiert drei Gruppen von Testfällen:

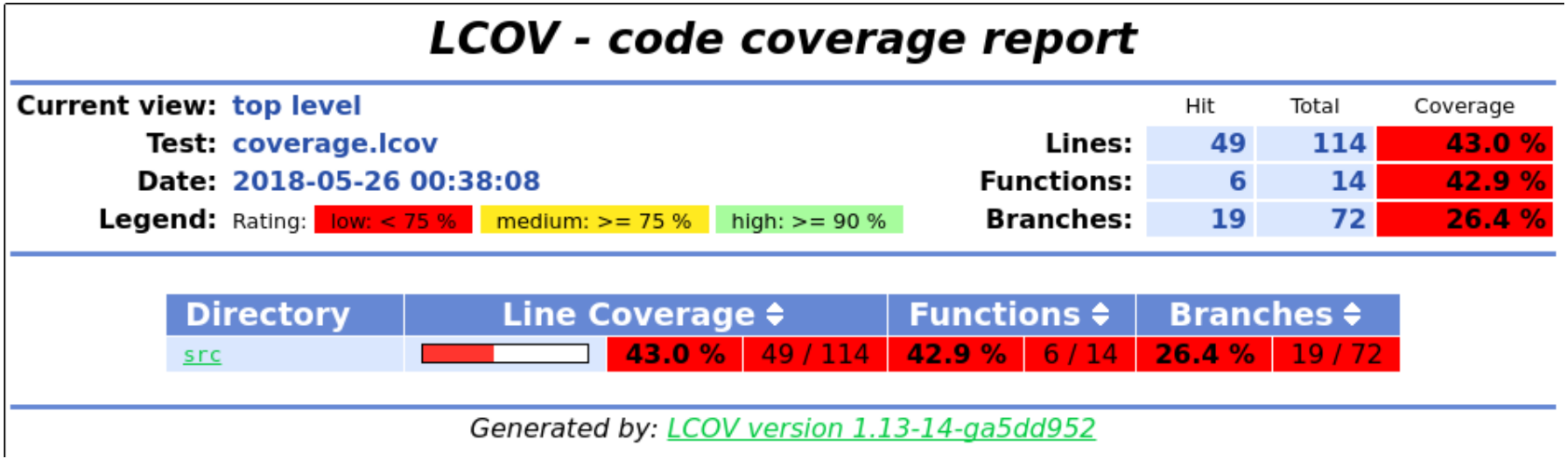
```
############## CONFIGURATION SECTION, add your testcases below
# Generelle Testfälle, sowohl für die eigene
# wie auch die fremde Implementierung
set(EZS_PQ_GENERAL_TESTS priority_queue_test1
                         priority_queue_test2)

# Mit dem AddressSanitizer inkompatible Tests
set(EZS_PQ_MALLOC_TESTS priority_queue_test_malloc)

# Testfälle ausschließlich für die
# eigene Implementierung
set(EZS_PQ_OWN_ONLY_TESTS "")
```

- Aktivieren eigener Tests: Eintrag in die entsprechende Liste

Codeüberdeckung: gcov/lcov



LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level		**Lines:**	49	114	43.0 %
Test:	coverage.lcov		**Functions:**	6	14	42.9 %
Date:	2018-05-26 00:38:08		**Branches:**	19	72	26.4 %
Legend:	Rating:	low: < 75 % / medium: >= 75 % / high: >= 90 %				

Directory	Line Coverage		Functions		Branches	
src		43.0 %	49 / 114	42.9 %	6 / 14	26.4 %	19 / 72

Generated by: LCOV version 1.13-14-ga5dd952

- Werkzeug aus der gcc-Toolchain
- Instrumentierung des Binärcodes $\rightsquigarrow$ *Laufzeitkosten*
- Protokollieren der Programmausführung
  - Wie oft wird jede Codezeile ausgeführt?
  - Welche Zeilen werden überhaupt ausgeführt?
  - Welche Verzweigungen wurden genommen?
- HTML Ausgabe: lcov

$\rightarrow$ Ziel: *vollständige Verzweigungsüberdeckung*!

Aufdecken von Laufzeitfehlern – AddressSanitizer

- *„Im besten Fall kracht es bei Speicherzugriffsfehlern!"*
- In Übungen: Verwendung von Clang AddressSanitizer [1][1]
- Checks zur Laufzeit
  - falsche Verwendung von Zeigern
  - nicht-definierte Integer-Operationen
  - Lesen uninitialisierten Speichers
  - Integer-Überlauf
  - ...

Entdeckt Fehler . . .

. . . nur, wenn die verwendeten Testfälle diese auslösen.

☞ **zur Laufzeit**

- Laufzeitkosten: ≈ 2 x

[1] `http://clang.llvm.org/docs/AddressSanitizer.html`

Clang AddressSanitizer – Verwendung

```
1  // program.cpp
2  int main(int argc, char **argv) {
3    int *array = new int[100];
4    delete[] array;
5    return array[argc];  // BOOM
6  }
```

```
$ clang++ -O1 -g -fsanitize=address program.cpp
```

```
$ ./a.out
```

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x602e0001fc64 at pc ...
```

- Wird von cmake-Skripten automatisch verwendet, wenn
  - Debugging aktiviert ist
  - und `clang` als Compiler verwendet wird
  - siehe `cmake/sanitizer.cmake`
- Aufruf von cmake

↝ `CC=clang CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Debug ..`

Statische Programmanalyse – Clang Static Analyzer

- Analyse des Quellcodes (C, C++, Objective-C)
- Keine Ausführung des Codes auf Hardware $\rightsquigarrow$ „statische Analyse”
- Eingabewerte als *symbolisch* angenommen
  $\rightarrow$ *symbolische Ausführung/Erreichbarkeitsanalyse*
- Verfügbare Checks[2]
  - Wertebereichsanalysen: Division mit Null
  - Verwendung uninitialisierter Variablen
  - ...
- Analyse ist *nicht fehlerfrei* (engl. sound)
  - Nicht möglich alle Fehler zu finden (engl. false negatives)
- Analyse ist *nicht präzise* (engl. precise)
  - Falsche positive Befunde sind möglich (engl. false positives)

[2] `http://clang-analyzer.llvm.org/available_checks.html`

Clang Static Analyzer – Verwendung

```
1  void test() {
2    int i, a[10];
       1 'i' declared without an initial value →
3    int x = a[i]; // warn: array subscript is undefined
       2 ← Array subscript is undefined
4  }
```

- Einzelne Datei überprüfen: `scan-build clang -c program.c`
- Übung: Aufruf von `scan-build` mit `cmake` als Argument

⇝ `CC=clang CXX=clang++ scan-build cmake ..`

⇝ `scan-build make`

- Fehler/Warnungen gefunden → Ausgabe von HTML Dateien
- Aufruf von `scan-view` wie in Ausgabe beschrieben

Literatur I

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov.
AddressSanitizer: A fast address sanity checker.
In *Proceedings of the USENIX Annual Technical Conference*, pages 309–318, 2012.