Tests mit dem Debugger, gdb+Python

UNIX-Stammtisch Dresden, 7.3.12

Reinhard Wobst

Inhaltsverzeichnis

1. Idee

Drei Ideen werden mit diesem Pythonskript *wbd* (für das noch ein griffiger Name gesucht wird) realisiert:

- Es wird ein Debugger genutzt, um interne Zustände während der normalen Ausführung zu testen.
- Gleichzeitig sollen möglichst bequem ein oder mehrere Sets von Breakpoints automatisiert gesetzt werden.
- Testanweisungen sollen in Kommentaren versteckt werden, d.h., der eigentliche Code wird durch das Nutzen des Skripts nicht verändert.

1.1. Nutzung

- Einmal für "**erweiterte Unittests**": So lange alle Bedingungen erfüllt sind, läuft das Programm ohne Änderung durch (nur durch den Debugger gebremst).

- Zum anderen als bequemes **Hilfsmittel zum Debuggen**, wobei komplexe Testfunktionen und mehrere Szenarien fest im Quellcode enthalten sind (im einfachsten Fall nutzt man mehrere feste Breakpoints, die auch beim Umstruktieren des Codes erhalten bleiben).

1.2. Arbeitsweise

Python

- Erzeugen von `.pdbrc` für den Python -Debugger `pdb` im aktuellen (Aufruf-)Verzeichnis, das eine Liste von Breakpoints, optional mit Bedingungen, enthält. Zusätzliche Funktionen können in einem Modul untergebracht werden, das beim Start anzugeben ist.

C/C++

- Für **gdb** wird ein temporäres File **.wbdgdb** mit entsprechenden Breakpoint-Beschreibungen erzeugt. Zusätzliche Python-Funktionen, die vom gdb-Python-Interface aus nutzbar sind, können im File ***AAA-gdb.py*** definiert werden, wobei ***AAA*** der Name des ausführbaren Programms ist.

2. Beispiele

2.1. Mehrere Breakpoints in Python oder C/C++ setzen

- **Vor** die entsprechenden Zeilen ist jeweils eine Zeilen im Quellcode einzufügen, die lediglich den Kommentar #DEB (Python) bzw. //DEB (C/C++) enthält. Die Zeile kann eingerückt sein.
- Das Programm ist wie üblich zu starten, nur muss vor die Kommandozeile noch der Skriptname "wbd" gesetzt werden.

Beispiel für Python

Die normale Kommandozeile laute

```
tt.py 40
```

In diesem Fall ist also zu tippen

```
wbd tt.py 40
```

Es erscheint eine Ausschrift etwa wie

```
Breakpoint 1 at /home/wobst/util/python/wbd/tt.py:9
> <string>(1)<module>()
(Pdb)
```

Nun ist das Programm wie üblich mit "c" zu starten und läuft unter der Regie des Python-Debuggers pdb.

Beispiel für C

Die normale Kommandozeile laute

```
prototyp 12
```

Man tippt

```
cwbd prototyp 12
```

(der Skriptname ist hier also **cwbd**) - und der Debugger **gdb** startet:

```
cwbd.py prototyp 12
GNU gdb (GDB) SUSE (7.1-3.12)
...
Breakpoint 1 at 0x4005f8: file prototyp.c,
line 15.
(gdb)
```

Hier muss der Debugger mit "r" gestartet werden.

2.2. Bedingte Breakpoints

Eine Bedingung wird nach dem Marker #DEB bzw. //DEB-Zeile als Kommentar in der Folgezeile angegeben, wobei diese mit "\" am Zeilenende fortgesetzt werden kann.

Beispiel für Python

```
//DEB
// a<3 or a==3 or \
// a == b
c = f(a)
```

Es sind alle Ausdrücke erlaubt, die der Debugger (pdb bzw. gdb) zulässt.

2.3. Breakpoint-Kommandos

Zusätzlich kann man für C/C++ (Tool cwbd) noch **nach** den (optionalen) Bedingungen Kommandos angeben, die auf den Marker "//CMD" folgen müssen.

Beispiel

```
//DEB
//r > 0.
//CMD
//python V("r"); V("x")
```

In diesem Beispiel ist V eine vordefinierte Python-Funktion, vgl. 2.5. In Python ist das nur über Modulfunktionen möglich, vgl. 2.5.

2.4. "White Box Test"

Der Sourcecode wird wie bei Breakpoint-Kommandos vorbereitet, nur hält der Debugger *nicht* an, wenn die Bedingung erfüllt ist. Der Marker muss in diesem Fall **TEST** heißen, nicht **DEB**, und der Schalter `-u` (wie *unit test*) ist beim Aufruf zu setzen:

File tt.py:

```
#TEST
# a > 0
...
```

Kommandozeile:

```
wbd -u tt.py 40
```

Für C/C++:

```
cwbd -u prototyp 12
```

2.5. Vordefinierte Funktionen

Es können Python-Funktionen vordefiniert werden, die man in Bedingungen und Kommandos von Breakpoints aufrufen kann.

Python

Die Funktionen müssen in einem Modul definiert sein, z.B. *M.py* mit dem Inhalt

```
def _p(s):
  print s, '***'
  return True
```

Der Kommentar vor einem Breakpoint kann dann so aussehen:

```
#DEB
# M._p(a) and a == 33
```

Dann ist **wbd** mit dem Schalter **-m** zu rufen:

```
wbd -m M tt.py 40
```

C/C++

Python-Funktionen, die zur Laufzeit mit **python** oder als Komfortfunktion (vgl. 5.2) zu rufen sind, müssen im File ***prototyp*-gdb.py** definiert sein, wenn ***prototyp*** das zu debuggende Programm ist (vgl. *info-Dokumentation zu gdb > Extending GDB > Python > Python-API > Auto-*

loading). Mehr dazu im Beispiel in 5.2. Der Schalter `-m` ist nur bei `wbd` zugelassen.

2.6. Mehrere Sets von Tests oder Breakpoints

Die Marker DEB/TEST können durch andere ersetzt werden, die nach dem Schalter *-t* (wie *token*) beim Aufruf anzugeben sind:

File tt.py:

```
#Special
# (a>>1) & 3
```

Kommandozeile:

```
wbd -t Special -u tt.py 37
```

2.7. Breakpoint-Kommandos

C/C++

Marker:

```
//CMD
```

In einer Zeile **nach** dem Marker. Optional kann in den Zeilen vorher eine Bedingung stehen; dann wird das Kommando nur bei erfüllter Bedingung ausgeführt.

Python

- Breakpoint-Kommandos können in *pdb* (und das auch erst ab Python Version 2.6) nur interaktiv eingegeben werden, nicht in *.pdbrc* hinterlegt.
- Ausweg: Im "globalen Modul" eine Funktion definieren, die das Kommando ausführt und *True* zurückgibt. Diese wird mit der Bedingung

mit *and* verknüpft. Je nachdem, ob die Funktion vor oder nach der Bedingung erscheint, wird das Kommando immer oder nur bedingt ausgeführt. Das ist sogar flexibler als bei *gdb*.

Beispiel: vgl. 2.5.

2.8. Quellfiles angeben

Standardmäßig werden alle Python/C/C++ Files im aktuellen Baum durchsucht, erkenntlich an den Endungen .py (Python) bzw. .c, .cc oder .cpp (C/C++).

Mit dem Schalter `-s` (*source*) kann der Name eines Files mit einer Liste der Quellfiles angegeben werden, jeweils einer pro Zeile. Ein führendes "./" ist zulässig.

3. Aufruf, Files, Voraussetzungen

Das Kommando `wbd -h` gibt folgenden Hilfetext aus:

```
wbd.py [options] commandline

Options:

-h, --help          show this help message and exit
-u, --utest         unittest modus: stop if breakpoint
                      condition is false;
                    default: debug modus, stop if breakpoint
                      condition is true
-t marker, --token=marker
                      name of the marker; default: DEB for
                       debug modus, TEST for unittest modus
-m FILE, --module=FILE
                      module containing extra definitions
                      available at breakpoints
-s FILE, --src=FILE   file with source file names,
```

```
                one per line; default:
              *.py in tree for wbd,
              *.c/*.cpp in tree for cwbd
```

Bei **cwbd** fehlt die Option **-m**.

4. Unterschied zu "assert" und normalem Debugging

4.1. Debugging

- ganze Klassen von optional bedingten Breakpoints können vorgegeben werden;
- automatisches Nachziehen bei jeder Quelltextumstellung;
- mögliche Kontrollausgaben, ohne dass in auszuführenden/compilierten Quelltextzeilen etwas verändert wird;
- nützlich, wenn die Fehlersuche schwierig wird und es eine Anzahl "verdächtiger Stellen" gibt, an denen Daten inspiziert werden müssen.

Nachteil

Bei zu vielen Breakpoints wird der Quelltext unlesbar.

4.2. Testen

assert-Anweisungen

Gegenüber *assert*-Anweisungen (die sich bei Python-Skripten verbieten, wenn deren Aufrufbedingungen nicht zu kontrollieren sind, z.B. in Plugins), hat *wbd/cwbd* zwei wichtige Vorteile:

- Über vordefinierte Funktionen ist eine Wechselwirkung zwischen einzelnen Stellen möglich, z.B. der Test auf die Aufrufreihenfolge von Funktionen oder Codestücken.
- Tests können mittels variabler Marker in Klassen eingeteilt werden, wobei immer nur eine Art Test ausgeführt wird (Schalter `-t`).
- In Abhängigkeit von Test können Breakpoints sogar manipuliert werden (zumindest bei **`gdb`** über das Python-Interface).

Nachteil für C/C++:

assert-Anweisungen erlauben alle Ausdrücke, die der Compiler in diesem Kontext versteht (z.B. die Interpretation von Preprozessor-Makros).

Unittests

- Zusätzlich kann der "innere Zustand" des Programms getestet werden, ohne extra auszuführenden oder zu compilierenden Testcode einzufügen.

Nachteil

Automatisierungsgrad von Unittests wird nicht erreicht - diese werden damit nicht überflüssig.

5. Weitere Beispiele

5.1. Nutzen des gdb-Python-Interfaces

Das C-Programm heiße *prototyp.c*; in *prototyp-gdb.py* sei die Funktion

```
def V(name):
    print "%s:" % name, \
      gdb.selected_frame().read_var(name)
    return True
```

definiert. Ein Breakpoint in *prototyp.c* der Gestalt

```
    //DEB
    //x == 12.
    //CMD
    //python V("r"); V("x")
```

führt etwa zu folgender Ausgabe nur für den Wert x = 12:

```
Breakpoint 1, main (argc=2,
      argv=0x7fffffffde38) at prototyp.c:16
16          printf("sqrt(%g) = %.15f\n", x, r);

r: 3.4641016151377548
x: 12
(gdb)
```

(vgl.a. *info gdb > Extending GDB > Python > Python-API > Frames in Python*).

5.2. "Tests mit Gedächtnis" in Python und C/C++

Python

Es soll die erwartete Reihenfolge von Rufen verschiedener Funktionen getestet werden. Im Beispiel werden Funktionen *fct1(), fct2(),* ... aus dem Modul *sub* gerufen. Erwartet wird, dass die Nummern der Funktionen nur ansteigen.

Das Modul **sub.py** habe folgende Struktur:

```
def fct1():
```

```
    global Sum
    #TEST
    #ordertest.order(1)
    Sum += 1

def fct2():
    global Sum
    #TEST
    #ordertest.order(2)
    Sum += 2
...
```

Das Modul **ordertest.py** (nur von *wbd* benutzt) habe den Inhalt

```
Order = [-1]

def order(i):
```

```
    global Order
    if i <= max(Order):
        return False
    Order.append(i)
    return True
```

Das Hauptprogramm **order.py** selbst sehe so aus:

```
import sub

for i in (1,2,5,8,13):
    exec ('sub.fct%d()' % i)

print "sum:", sub.Sum

sub.fct11()
```

Die Kommandozeile (Unittest-Modus)

```
wbd -u -m ordertest order.py
```

liefert die Ausgabe

```
...
(Pdb) c
sum: 29
>
/home/wobst/util/python/wbd/sub.py(68)fct11()
-> Sum += 11
(Pdb)
```

Das ist korrekt: Die Funktion order() testet auf steigende "Funktionsnummern", was bis zur Ausgabe von *sum* der Fall ist. Beim Ruf *sub.func11()* wird das Prinzip verletzt, und der Debugger hält an dieser Stelle an.

C/C++

Im Beispiel wird eine Wurzel iterativ berechnet. Es wird geprüft, ob die Defekte immer kleiner werden. Die zu testende Funktion in *prototyp.c* sieht so aus:

```
double root(double x)
  {
   int cnt = 0;
   double w = x/2.;
   double def;

   while((def = fabs(w*w - x)) > x*1.e-14)
   {
    //TEST
    //$smaller(w*w-x)
    printf("step %d:\tw = %.15g,
```

```
      def=%g\n", ++cnt, w, def);
    w = (w + x/w)/2.;
  }

  return w;
}
```

Zum Ausdruck "*w*w-x*": *gdb* kann einfache Ausdrücke berechnen, aber ein "*fabs(w*w-x)*" würde einen Fehler erzeugen, denn C-Bibliotheken können zur Laufzeit nicht dynamisch eingebunden werden.

Die sog. Komfortfunktion **`smaller()`** wird im Python-File ***`prototyp-gdb.py`*** definiert, vgl. *info-Dokumentation zu gdb > Extending GDB > Python > Python-API > Functions In Python:*

```
import gdb

class Pname(gdb.Function):
```

```
    def __init__(self):
        super(Pname, self).__init__("smaller")
        self.last = None

    def invoke(self, x):
        x = abs(x)
        if not self.last or self.last >= x:
            self.last = x
            return True
        print "*** increment: last = %s, \
                  new = %s" % self.last, x)
        return False

Pname()
```

Ein Lauf mit der Zeile

```
cwbd -u prototyp 12
```

(Start mit **gdb**-Kommando "*r*") läuft nun glatt durch. Wird jedoch in der Zeile **double w = x/2.** das **x/2.** durch 1. ersetzt, so vergrößert sich der Defekt nach dem ersten Schritt, und er erscheint

```
step 1: w = 1, def=11
*** increment: last = 11, new = 30.25

Breakpoint 1, root (x=12) at prototyp.c:30
30   printf("step %d:\tw = %.15g, def=%g\n", \
           ++cnt, w, def);
(gdb)
```

→ weitaus intelligentere Tests möglich!

6. gdb und Python-API

Vgl. dazu

info-Dokumentation zu gdb > Extending GDB > Python > Python-API > Functions In Python

sowie

info-Dokumentation zu gdb > Extending GDB > Python > Python-API > Breakpoints in Python

6.1. Komfortfunktionen

Diese werden mit *$name(...)* gerufen und sind wie in Beispiel 5.2 zu definieren:

- Der Konstruktor __init__() legt den Namen fest
- invoke() wird beim Aufruf der Funktion zur Laufzeit ausgeführt

- abschließend ist die Klasse zu instanziieren

6.2. gdb-Kommandos ausführen

Mit der Funktion

```
gdb.execute(command_as_string)
```

- vgl. 6.3.

6.3. Breakpoints verwalten

- Breakpoint erzeugen:

```
B = gdb.Breakpoint("prototyp.c:31")
```

- Breakpoint bedingt löschen:

```
def cond_del():
  try:
    if B.hit_count >= 2:
      print "*** breakpoint %d deleted" % \
        B.number
      gdb.execute("delete %d" % B.number)
  except:
    pass
  return True
```

(geht natürlich auch über Bedingung per Python-Funktion)

6.4. Stackframes und Symbole analysieren

vgl. Funktion von oben

```
def V(name):
    print "%s:" % name, \
      gdb.selected_frame().read_var(name)
    return True
```

Ebenso kann man aus der Symboltabelle Name, Typ, Block, Adressklasse u.a. ermitteln.

Weiteres vgl. Dokumentation (bis hin zum Ermitteln und Setzen der Thread-ID).

7. Links

http://sourceware.org/gdb/wiki/PythonGdbTutorial

... und zum Abschluss: Sag' niemals "nie"! ...

4 GB Daten auf Lochkarten (1959)

Die doppelte Datenmenge auf einem USB-Stick (2012)