Symbolic Debugging

SS 2007

Johannes Kepler University

Linz, Austria

Dr. Toni Jussila
Institut for Formal Models and Verification

http://fmv.jku.at/kv

Symbolic Debugging

- Use an *external observation tool* to analyze program state ie.
  - commands to be executed (program counter) and
  - program data.
- term *symbolic* refers to the fact that the tool operates on source level (as opposed to a machine-level debugger)

Benefits over `printf`-debugging (Zeller):

1. getting started fast
2. flexible observation
3. transient sessions

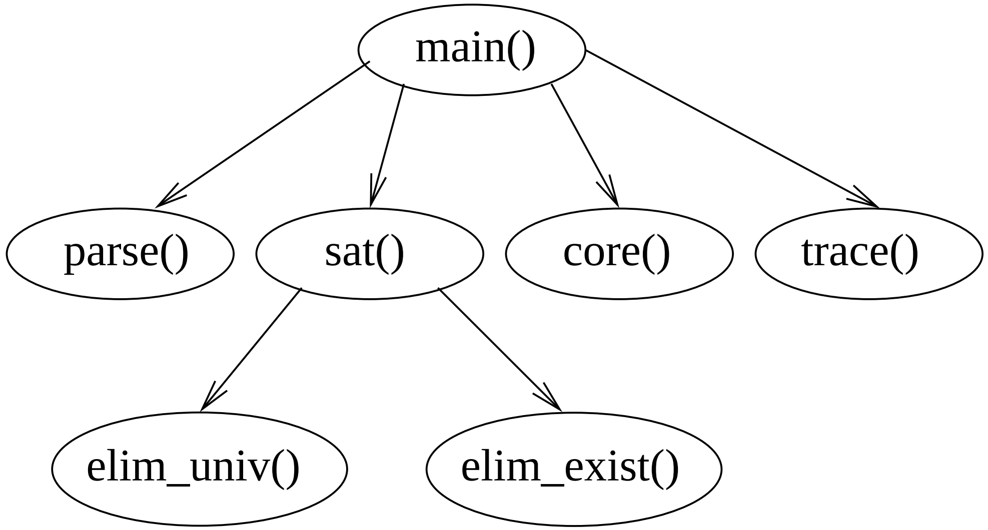
GNU Debugger `gdb`

- initially written by Richard Stallman 1986

- interactive program controlled via a command line
- supports: C, C++, Fortran, Ada, Modula-2
- works on both source level and machine code level
- machine code support for many processors
- `http://en.wikipedia.org/wiki/GNU_Debugger`

Case study: `ebddres`

- `ebddres` is a BDD based QBF solver producing refutation proofs



- we will study a failure in procedure `init_buckets()`

- also a good test case for delta debugging

- important data structures: `Var`, `Clause`, `And`, `Or`, `Ele`

Starting gdb

- executables need to be instrumented with *debugging information*
  - locations, names and types of variables and functions
  - correspondence between source lines and machine code addresses
- GNU Compiler Collection (`gcc`) produces this with switch `-g`

```
$ gcc -g -o sample sample.c
```

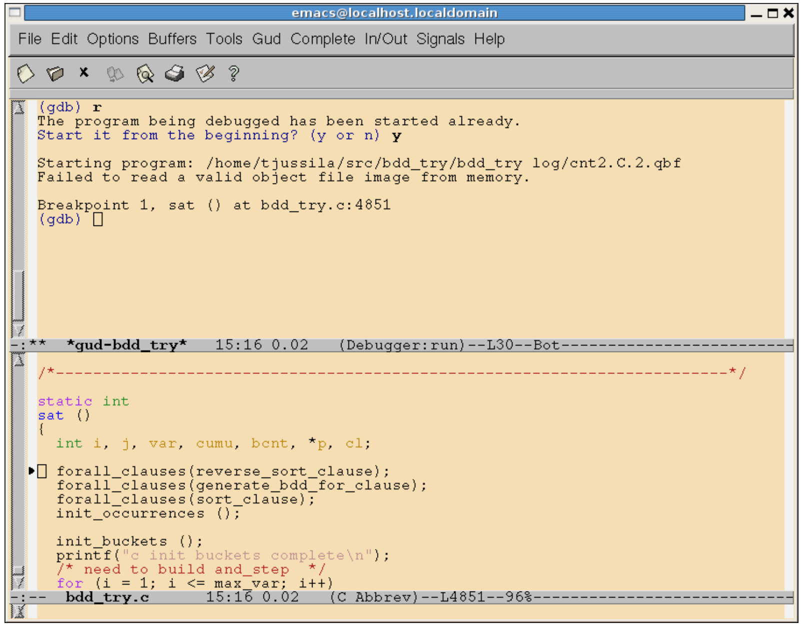
- then just start `gdb` with the executable

```
$ gdb sample
GNU gdb 6.1, Copyright 2004 Free Software Foundation, Inc.
(gdb) _
```

With GNU Emacs

GNU Emacs provides integrated support for using `gdb`.

- started with `M-x gdb` where `M` is the Meta character
- all input and output of `gdb` goes through an Emacs buffer
- view and edit source files while debugging



- keyboard shortcuts for common `gdb` commands
  - `C-c C-s` is `gdb` command step, with argument `M-5 C-c C-s`
  - `C-c C-n` is `gdb` command next

General commands

- `info` displays information of the program being debugged
  - `info warranty; info all-registers;`
- `show` displays information of the debugger
  - `show charset; show architecture;`
- `help` followed by a command displays its purpose and usage
- `set` set variable value to an expression
  - `set args a1 a2; set i=j+k;`
- `quit`

Stopping and Continuing

- breakpoint: execution stops whenever a certain location (function, source code line) is reached (command `break`)
- watchpoint: stop whenever the value of some expression changes (command `watch`)
- catchpoint: stop whenever a special event occurs (C++ exception, loading a dynamic library, catching a signal) (command `catch`)
- when a program stops, the called functions (stack frames) can be displayed by printing a back trace (command `bt`)
- `info {breakpoints,watchpoints}`
- `list`, prints source lines at the breakpoint

Stopping and Continuing II

Traverse source program line by line:

- command `step count`, execute `count` line(s), follow function calls
  - library functions do not have debugging information
- command `next count`, execute `count` line(s), do not follow function calls

Resuming execution:

- command `continue`, abbr. `c`, continues execution
- command `finish`, continues execution until this frame finishes
- command `until`, abbr. `u`, continues execution until a loop finishes

Stopping and Continuing III

- break-, watch-, and catchpoints can be augmented with
  1. ignorecount: `ignore bnum count`
  2. condition: `condition bnum expression`
  3. a command list: `command bnum ...`
- Ignorecount is a natural number that is decremented each time a breakpoint is reached. Break only if zero (default).
- Condition is an expression. Break only if this expression non-zero.
- Command list allows for instance data values be printed each time a breakpoint triggers.

Deleting Breakpoints

- command `delete bnum`, abbr. `d bnum`
  - without argument deletes all breakpoints
- alternative, command `clear`
  - argument line number or function
- if you think you will need the breakpoint in future, use command `disable bnum`
- to reactivate, use command `enable`
  - variant `enable bnum once` or `enable bnum delete`

Examining Data

- command `print` abbr. `p`, prints expressions of your source language
- command `x/nfu`, examines memory at a lower level
  - `n`, repeat count; `f`, display format; `u`, unit size
  - example: `x/3uh 0x54320`, three halfwords (`h`) of memory as unsigned decimal integers (`u`).
- command `display`, abbr. `d`, causes a value of an expression to be printed whenever program stops
- command `whatis`, shows the data type of a variable
- command `ptype`, prints the detailed type of variable

Convenience Variables

- for complex data access, you may need new variables to store values to
- `gdb` supports this by *convenience variables*, prefixed by `$`
- several defined internally, like:
  - $pc, the value of the program counter (`x/i $pc`)
  - $sp, the value of the stack pointer
  - $eax, internal register
  - `show convenience; info registers`

Examining Program Code

- command `list` displays source lines
  - `list 1000`, print `listsize` lines starting from line 1000
  - `list main`, print `listsize` lines starting from function main
  - `list +`, print lines just after the lines last printed
- `set listsize` allows to modify number of lines
- `show listsize` prints its current value
- `edit line` allows modifying source lines, default editor `/bin/ex`
  - more practical under GNU Emacs

At Machine Level

- command `info line linenumber`
  - show the start and end addresses of the machine code of `linenumber`
- conversely, `info line addr` shows which source line covers address addr
- command `disassemble`, abbr. `disas`, shows the assembly code from a given address range
- command `set dissassembly-flavor intel`

Case Study: `gcd`

Source code:

```
int gcd(int a, int b)
{
  int t;
  while (b != 0)
    {
      t = b;
      b = a % b;
      a = t;
    }
  return a;
}
```

Under `gdb` command `disas gcd` gives:

```
<gcd+0>:  push   ebp
<gcd+1>:  mov    ebp,esp
<gcd+3>:  sub    esp,0x10
<gcd+6>:  jmp    <gcd+30>
<gcd+8>:  mov    eax,DWORD PTR [ebp+12]
<gcd+11>: mov    DWORD PTR [ebp-4],eax
<gcd+14>: mov    eax,DWORD PTR [ebp+8]
<gcd+17>: cdq
<gcd+18>: idiv   DWORD PTR [ebp+12]
<gcd+21>: mov    DWORD PTR [ebp+12],edx
<gcd+24>: mov    eax,DWORD PTR [ebp-4]
<gcd+27>: mov    DWORD PTR [ebp+8],eax
<gcd+30>: cmp    DWORD PTR [ebp+12],0x0
<gcd+34>: jne    <gcd+8>
<gcd+36>: mov    eax,DWORD PTR [ebp+8]
<gcd+39>: leave
<gcd+40>: ret
```

Postmortem Debugging

Several operating systems can be set up to allow dumping of a *core file*. This file is created if the program crashes and contains the programs memory state.

- `ulimit -c` shows the maximum size of core file
- `gdb` supports core files

```
$ gdb ./bdd_try core
...
Core was generated by `./bdd_try /qbf/adder-2-sat.qdimacs'.
Program terminated with signal 11, Segmentation fault.
#0  0x08050ceb in init_buckets () at bdd_try.c:4334
4334              cl = clauses + var->occurrences[j];
```

What is missing?

Given a point, where your program fails, why can you not go backwards ?

- memory reasons, you would have to remember each program state (at machine code level), call stack etc.

Inserting a breakpoint to point of failure may lead to a tedious session; the breakpoint may trigger arbitrarily many times before failure reached.

- a breakpoint can be instrumented with a command sequence. Let this sequence be `continue`.

- rerunning the program causes failure. Then, however, `info break` tells how many times your breakpoint triggered.

- This gives you the new ignore count. Use binary search to determine where program state was corrupted.

User-defined Commands

It is possible to define your own command sequences and document them:

```
define adder
  print $arg0 + $arg1 + $arg2
end
```

- `define command` to define functions
- `document command` to write their documentation (shown by `help command`)