DEBUGGING: OBSERVING AND TRACKING

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification
Observing a Program

- deduction tells what might happen
- observation tells what is actually happening

Observation at a glance:

- collect facts about what has happened in a concrete run
- look into actual program execution
- approaches:
  - logging
  - interactive debugging
  - post-mortem debugging
  - summarization techniques
Principles of Observation

- **Do not interfere.** Observation should be effect of original run, not caused by observation mechanisms.

- **Know what and when to observe.**
  - which part of the state
  - at which moments during execution

- **Proceed systematically.** Guide the search by scientific method, not by random.
printf Debugging

simplest (and probably most widespread) way of debugging: insertion of printf statements into the code for learning about the values of variables

drawbacks:

- cluttered code
  - do not contribute to understand the code in general
  - have to be removed after the debugging

- cluttered output
  - often a huge mass
  - interleaving with ordinary output

- slowdown

- loss of data in case of crash
Desired Properties of Logging Techniques

- standard formats:
  - search and filter for specific
    - code locations
    - events
    - data

- variable granularity
  - sharpens focus
  - improves performance

- disabling feature
- persistence feature
Customizing Logging

- **simple possibility:** `dprintf (...)`
  - same behavior as `printf (...)`, but
    - ... write to a special debugging log
    - ... allow output to be turned off
    - ... prefix with information like the date or a marker, e.g.,
      `DEBUG: size = 3`
  - drawback: performance if called often

- **more cost effective:** use a logging macro
  - easy to turn off (e.g., at compile time)
  - may involve expensive calculations
  - may contain information about their own location
Logging Frameworks

- general purpose libraries for logging are available
- standardize the process of logging
- main components
  - logger: collecting message and metadata to be logged
  - formatter: aligning collected information for output, e.g., convert objects into strings
  - handler (appender): display our write output
- severity levels: FATAL, ERROR, WARNING, INFO, DEBUG, TRACE
Logging with Aspects

aspect-oriented programming: separate cross-cutting concerns into individual syntactic entities (aspects)

basic concepts

- advice
- cutpoints
- joinpoints

⇒ logging and actual computation are not intertwined
Example: Logging with Aspects

log entry and exit of method \texttt{buy} defined in class \texttt{Article}

```java
public aspect LogBuy {
    pointcut buyMethod():
        call(public void Article.buy());
    before(): buyMethod() {
        System.out.println("Entering Article.buy()")
    }
    after(): buyMethod() {
        System.out.println("Leaving Article.buy()")
    }
}
```
Debugger

drawbacks of logging approach:

■ writing and integrating code
■ rebuild and rerun program

⇒ use external observation tool (debugger) that

■ observe program states
■ stop program at a certain state
■ manipulates program states
■ does not change original code
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```
```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

### Preparation:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hypothesis</strong></td>
<td>input “11 14” works</td>
</tr>
<tr>
<td><strong>prediction</strong></td>
<td>output is “11 14”</td>
</tr>
<tr>
<td><strong>experiment</strong></td>
<td>run with input “11 14”</td>
</tr>
<tr>
<td><strong>observation</strong></td>
<td>output is “0 11”</td>
</tr>
<tr>
<td><strong>conclusion</strong></td>
<td>hypothesis rejected</td>
</tr>
</tbody>
</table>
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
1 int *a;
2 int i;

3 a = (int *)malloc((argc - 1) * sizeof(int));
4 for (i = 0; i < argc - 1; i++)
5 a[i] = atoi(argv[i + 1]);
6 shell_sort(a, argc);
7 printf("Output: ");
8 for (i = 0; i < argc - 1; i++)
9 printf("%d ", a[i]);
10 printf("\n");
11 free(a);
12 return 0;
}
```

```
$ gcc -g -o shell shell.c
$ ./shell 11 14
Output: 0 11
$ gdb shell
GNU gdb (Ubuntu 7.11.1-0ubuntu1 16.5) 7.11.1
(gdb)
10 return 0;
```

<table>
<thead>
<tr>
<th>Preparation:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypothesis</td>
<td>input “11 14” works</td>
</tr>
<tr>
<td>Prediction</td>
<td>output is “11 14”</td>
</tr>
<tr>
<td>Experiment</td>
<td>run with input “11 14”</td>
</tr>
<tr>
<td>Observation</td>
<td>output is “0 11”</td>
</tr>
<tr>
<td>Conclusion</td>
<td>hypothesis rejected</td>
</tr>
</tbody>
</table>
```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

**Hypothesis 1:**

| hypothesis | a[0] becomes zero |
| prediction | a[0] = 0 in line 9 |
| experiment | observe a[0] |
| observation | a[0] = 0 |
| conclusion | hypothesis confirmed |

---

**Example: Broken Shell Short**

```c
Example: Broken Shell Short

```
int main (int argc, char *argv []) {
    1    int *a;
    2    int i;

    (gdb) break 7
    Haltepunkt 1 at 0x4007e6: file shell.c, line 7.
    (gdb) run 11 14
    Starting program: shell 11 14
    Breakpoint 1, main (argc=3, argv=0x7fffffffdd98) at shell.c:7
    7 for (i = 0; i < argc - 1; i++)
    (gdb) print a[0]
    $1 = 0

    return 0;
}

Example: Broken Shell Short
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Hypothesis 2:

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>infection in shell_sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>prediction</td>
<td>a = [11, 14], size = 2 in line 5</td>
</tr>
<tr>
<td>experiment</td>
<td>observe a, size</td>
</tr>
<tr>
<td>observation</td>
<td>a = [11, 14, 0], size = 3</td>
</tr>
<tr>
<td>conclusion</td>
<td>hypothesis rejected</td>
</tr>
</tbody>
</table>
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
    shell_sort(a, argc);
    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

Hypothesis 2:

hypothesis in shell_sort prediction a = [11, 14], size = 2 in line 5

observation a = [11, 14, 0], size = 3

conclusion hypothesis rejected

(gdb) break shell_sort
Breakpoint 2, shell_sort (a=0x602010, size=3) at shell.c:5
5 int h = 1;
(gdb) print a[0]
$2 = 11
(gdb) print a[1]
$3 = 14
(gdb) print a[2]
$4 = 0

Conclusion: Hypothesis Rejected
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Hypothesis 3:

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>size = 3 causes failure in shell_sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>prediction</td>
<td>if we set size = 2 program works</td>
</tr>
<tr>
<td>experiment</td>
<td>set size = 2</td>
</tr>
<tr>
<td>observation</td>
<td>as predicted</td>
</tr>
<tr>
<td>conclusion</td>
<td>hypothesis confirmed</td>
</tr>
</tbody>
</table>
int main (int argc, char *argv []) {
  1 int *a;
  2 int i;

  3 a = (int *)malloc((argc - 1) * sizeof(int));
  4 for (i = 0; i < argc - 1; i++)
    5 a[i] = atoi(argv[i + 1]);
  6 shell_sort(a, argc);

  7 printf("Output: ");
  8 for (i = 0; i < argc - 1; i++)
    9 printf("%d ", a[i]);
  10 printf("\n");

  11 free(a);
  12 return 0;
}

Hypothesis 3:

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>size = 3 causes failure in shell_sort</td>
<td>if we set size = 2 program works</td>
</tr>
<tr>
<td>experiment</td>
<td>set size = 2</td>
</tr>
</tbody>
</table>

**observation**

as predicted

**conclusion**

hypothesis confirmed
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
1    int *a;
2    int i;

3    a = (int *)malloc((argc - 1) * sizeof(int));
4    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

5    shell_sort(a, argc);

6    printf("Output: ");
7    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
8    printf("\n");

9    free(a);
10   return 0;
}
```

Hypothesis 4:

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>using <code>argc</code> instead of <code>argc-1</code> in <code>shell_sort</code> causes failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>prediction</td>
<td>output is “11 14”</td>
</tr>
<tr>
<td>experiment</td>
<td>change <code>argc</code> to <code>argc-1</code> in line 5</td>
</tr>
<tr>
<td>observation</td>
<td>as predicted</td>
</tr>
<tr>
<td>conclusion</td>
<td>hypothesis confirmed</td>
</tr>
</tbody>
</table>
Example: Broken Shell Short

```c
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

**Hypothesis 4:**

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hypothesis</strong></td>
<td>using <code>argc</code> instead of <code>argc-1</code> in <code>shell_sort</code> causes failure</td>
</tr>
<tr>
<td><strong>prediction</strong></td>
<td>output is “11 14”</td>
</tr>
<tr>
<td><strong>experiment</strong></td>
<td>change <code>argc</code> to <code>argc-1</code> in line 5</td>
</tr>
<tr>
<td><strong>observation</strong></td>
<td>as predicted</td>
</tr>
<tr>
<td><strong>conclusion</strong></td>
<td>hypothesis confirmed</td>
</tr>
</tbody>
</table>
Debugging: Summary

**important concepts** (selection):

- **breakpoint**
  (stop execution at certain line)
  (gdb) break 8

- **watchpoint**
  (stop execution when value of expression changes)
  (gdb) watch a[0]

- **conditional breakpoint**
  (stop execution at a specific location if condition is true)
  (gdb) break 8 if (a[0] == 0)

**benefits:**

- no modification of code
- flexible observation
- transient sessions
Automating Observations

- challenges in observing a program:
  - huge amount of states and events
  - new run $\rightarrow$ new observation
  - judging if a state is sane or not

- observation alone is not enough for debugging

- essential: compare observed facts with expected behavior

$\Rightarrow$ **assertions**: take small probes in state and time
Assertions (1/2)

An assertion is a Boolean expression at a specific point in a program which will be true unless there is a defect.

**Example:**

```c
assert(0 <= index && index < length);
```

- **Goal:** notify a programmer about a problem
- **Provides diagnostic information**
- **Easy to remove by recompilation,** e.g., defining the `NDEBUG` macro in C
- **Powerful in combination with fuzzing**
**Assertions (2/2)**

```python
assert (expr) asserts that an expression is true. The expression `expr` may or may not be evaluated.
```

- If the expression is true, execution continues normally.
- If the expression is false, what happens is undefined.


handling failed assertions

- terminate the program
- provide some message and continue
- throw an exception
- ask the user how to continue
Assertions: Pros and Cons

benefits:

- support better testing and easier debugging
  - detect very subtle problems
  - detect problems sooner after they occurred
- scalability and persistence
- executable comments about preconditions, postcondition, and invariants
- first step towards a formal spec

drawbacks:

- slow down of code
- usually if not executed, then little information gain (except on control flow)
- improper use can make programs incorrect
  - tempting to be used for error handling
Origins of Assertions

■ **safety check** of (intermediate) calculations
  (often checking the result is easier than obtaining it)

■ **precondition**:  
  - assert something that has to be true for code to execute
  - documents requirements
  - useful for failure diagnosis

■ **postcondition**: easy to check guarantee

■ **invariant**: property that has to hold during the whole program execution
  
  example: for a doubly-linked list it holds:
  ```c
  assert (n->next->prev == n);
  ```

■ **specifications**: conditions that the program should fulfill
Examples for Using Assertions

- stating that an argument of a function should not be null
  ```
  int div (int x, int y) {
    assert (y != 0);
    ...
  }
  ```
- checking the control flow
  ```
  switch (x) {
    case 1: ...; break;
    case 2: ...; break;
    case 3: ...; break;
    default: assert (0);
  }
  ```
- checking of representations
  ```
  assert (valid_structure (tree));
  ```
- array indices within bounds
- cached values are not out of dates
- ...

Some Pitfalls

- **defects in assertions**
  - reporting errors where none exists
  - reporting no error where an error exists
  - side-effects
    
    ```
    assert (x = 7);
    ```

- **misuse for error handling**
  
  ```
  int result = open (filename);
  assert (result != -1);
  ```

- **vacuous assertions**
  
  ```
  if (x) {
      y = 1;
  } else {
      y = 2;
  }
  assert (y == 1 || y == 2);
  ```