DEBUGGING: STRUCTURED DEBUGGING

WS 2018/2019

Martina Seidl
Institute for Formal Models and Verification
Cause of a Failure

The cause of any event ("effect") is a preceding event without which the effect would not have occurred.

to prove causality, one must show that

- the effect occurs when the cause occurs
- the effect does not occur when the cause does not.

advantages in programming

- programs are (high-level) abstractions of reality
- program runs are usually repeatable
- testing can be automated
Debugging: Ad-Hoc Approach

guess the cause of a failure based on

- intuition
- experience

problems with this approach

- a priori knowledge is necessary
- hardly systematic
- hardly reproducible
- hard to teach

**challenge**: systematically find an explanation for a failure
Debugging: Scientific Method

process of obtaining a theory that explains some aspects of the universe

process outline:

1. observe a failure
2. establish a hypothesis that is consistent with observations
3. make predictions based on the hypothesis
4. test the hypothesis by experiments and further observations
   - refine hypothesis if experiment satisfy the predictions
   - otherwise, create alternative hypothesis
5. repeat 3. and 4. until no refinement is possible
Example: Broken Shell Short

```c
1 void shell_sort (int a [], int size) {
2     ...
3 }
4
5 int main (int argc, char *argv []) {
6     int *a; int i;
7
8     a = (int *)malloc((argc - 1) * sizeof(int));
9     for (i = 0; i < argc - 1; i++)
10         a[i] = atoi(argv[i + 1]);
11
12     shell_sort(a, argc);
13
14     printf("Output: ");
15     for (i = 0; i < argc - 1; i++)
16         printf("%d ", a[i]);
17     printf("\n");
18
19     free(a);
20     return 0;
```
Example: Broken Shell Short

```c
1 void shell_sort (int a [], int size) {
    ...
}

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

**Preparation:**

<table>
<thead>
<tr>
<th><strong>hypothesis</strong></th>
<th>input “11 14” works</th>
</tr>
</thead>
<tbody>
<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
1 void shell_sort (int a [], int size) {
   ...  
}

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

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

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

11   free(a);
12   return 0;
```

**Hypothesis 1:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hypothesis</strong></td>
<td>a[0] becomes zero</td>
</tr>
<tr>
<td><strong>prediction</strong></td>
<td>a[0] = 0 before line 6</td>
</tr>
<tr>
<td><strong>experiment</strong></td>
<td>observe a[0]</td>
</tr>
<tr>
<td><strong>observation</strong></td>
<td>a[0] = 0</td>
</tr>
<tr>
<td><strong>conclusion</strong></td>
<td>hypothesis confirmed</td>
</tr>
</tbody>
</table>


Example: Broken Shell Short

```c
1 void shell_sort (int a [], int size) {
    ...
}

t void main (int argc, char *argv []) {
2 int *a; 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 2:

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Infection in <code>shell_sort</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>a = [11, 14], size = 2 after line 1</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>

JMU
Example: Broken Shell Short

```
1 void shell_sort (int a [], int size) {
    ...
}

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

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

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

9    free(a);
10   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>
Example: Broken Shell Short

```c
1 void shell_sort (int a [], int size) {
2     ...
3 }
4 int main (int argc, char *argv []) {
5     int *a; int i;
6     a = (int *)malloc((argc - 1) * sizeof(int));
7     for (i = 0; i < argc - 1; i++)
8         a[i] = atoi(argv[i + 1]);
9     shell_sort(a, argc);
10    printf("Output: ");
11    for (i = 0; i < argc - 1; i++)
12        printf("%d ", a[i]);
13    printf("\n");
14    free(a);
15    return 0;
```

Hypothesis 4:

<table>
<thead>
<tr>
<th>hypothesis</th>
<th>using argc instead of argc-1 in shell_sort causes failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>prediction</td>
<td>output is “11 14”</td>
</tr>
<tr>
<td>experiment</td>
<td>change argc to argc-1 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>
Summary: Scientific Method

- problem
- code
- failing run
- other runs

hypothesis

- hypothesis rejected: create new hypothesis
- hypothesis confirmed: refine hypothesis

prediction

experiment

observation + conclusion

diagnosis → fix
Deriving a Hypothesis

- **problem description**: what is going wrong?
- **program code**: if not available, work around defects
- **failing run**: execute the code and reproduce the problem; observe actual facts about the concrete run
- **alternate runs**: identification of anomalies — differences between failing run and passing runs
- **earlier hypotheses**:
  - include passed hypotheses
  - exclude failed hypotheses
Theories in Debugging

When the hypothesis explains all experiments and observations, the hypothesis becomes a theory.

A theory is a hypothesis that

- explains earlier observations
- predicts further observations

Context of debugging: A theory is called a diagnosis

This contrasts popular usage, where a theory is a vague guess.
Algorithmic Debugging

**basic idea:** (partially) automate the debugging process by interactively querying the user about infection sources

**approach:**

1. assume an incorrect result $R$ with origins $O_1, O_2, \ldots, O_n$
2. for each $O_i$, check whether $O_i$ is correct
3. if some $O_i$ is incorrect, continue at Step 1 with $R = O_i$
4. otherwise (all $O_i$ are correct), we find the defect in the computation of $R$ using the $O_i$
Example: Algorithmic Debugging

def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)

def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))

sort ([2,1,3]) = [3,1,2]

sort ([1,3]) = [3,1]
insert (2, [3,1]) = [3,1,2]

sort ([3]) = [3]
insert (1, [3]) = [3,1]
Example: Algorithmic Debugging

```python
def insert(elem, list):
    if len(list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert(elem, tail)

def sort(list):
    if len(list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert(head, sort(tail))
```

sort([2,1,3]) = [3,1,2]  
insert(2, [3,1]) = [3,1,2]  
sort([1,3]) = [3,1]  
insert(1, [3]) = [3,1]  
sort([3]) = [3]
Example: Algorithmic Debugging

def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)

def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))

sort ([2,1,3]) = [3,1,2]
insert (2, [3,1]) = [3,1,2]

sort ([1,3]) = [3,1]
insert (1, [3]) = [3,1]

sort ([3]) = [3]
Example: Algorithmic Debugging

def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)

def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))

sort ([2,1,3]) = [3,1,2]

insert (2, [3,1]) = [3,1,2]

sort ([1,3]) = [3,1]

insert (1, [3]) = [3,1]

sort ([3]) = [3]
Example: Algorithmic Debugging

def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)

def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))

sort ([2,1,3]) = [3,1,2]

sort ([1,3]) = [3,1]

insert (2, [3,1]) = [3,1,2]

sort ([3]) = [3]

insert (1, [3]) = [3,1]
Algorithmic Debugging: Critical Discussion

- drive the search for a defect in a systematic way guided by human input

- problems on real-world scenarios:
  - scalability: number of functions, shared data structures, ...
    ⇒ works best for functional and logical programming languages
  - process is too mechanical: programmer has to assist the tool

⇒ replace programmer by oracle that knows the external specification of the program
not every problem needs the strength of a the scientific method, but for complex problems it is useful to

- be explicit is important to understand (and find) the problem
- write down hypotheses and observations in order to know
  - where you are
  - where you have been
  - where you want to go
  - what you want to get
Reasoning about Programs for Debugging

- **deduction** (0 runs)
  
  reason from (abstract) program code to concrete runs

  ⇒ static analysis

- **observation** (1 run)
  
  inspection of a single program run

  ⇒ facts about program execution

- **induction** (n runs)
  
  reasoning from the particular to the general

  ⇒ summary of findings from multiple runs

- **experimentation** (n controlled runs)
  
  refinement and rejection of hypotheses

  ⇒ scientific method