Asserting Expectations
Andreas Zeller
Search in Time

• During execution, the state becomes infected.

• Basic idea: Observe a transition from sane to infected.
Manual Observation
Automated Observation
Automated Observation

what to observe

when to observe

what to expect
Basic Assertions

if (divisor == 0) {
printf("Division by zero!"); 
abort();
}

Specific Assertions

assert (divisor != 0);
Implementation

void assert (int x)
{
    if (!x)
    {
        printf("Assertion failed!\n");
        abort();
    }
}
Execution

$ my-program
Assertion failed!
Abort (core dumped)
$

Better Diagnostics

$ my-program
divide.c:37:
    assertion ‘divisor != 0’ failed
Abort (core dumped)
$ _
Assertions as Macros

```c
#ifndef NDEBUG
#define assert(ex) \((ex) ? 1 : (cerr << __FILE__ << ":" << __LINE__ << ": assertion \"" #ex "\" failed\n", abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```
Automated Observation

<table>
<thead>
<tr>
<th>what to observe</th>
<th>when to observe</th>
<th>what to expect</th>
</tr>
</thead>
<tbody>
<tr>
<td>state checked in assertion</td>
<td>location of assertion</td>
<td>checked property of program state</td>
</tr>
</tbody>
</table>
When to observe

- Data invariants
- Pre- and postconditions
Asserting Invariants
A Time Class

class Time {
public:
    int hour();    // 0..23
    int minutes(); // 0..59
    int seconds(); // 0..60 (incl. leap seconds)

    void set_hour(int h);
    ...
}

Any time from 00:00:00 to 23:59:60 is valid
Ensuring Sanity

void Time::set_hour(int h)
{
    // precondition
    assert (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);

    ...

    // postcondition
    assert (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
}
Ensuring Sanity

```cpp
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());  // precondition

    ...

    assert (sane());  // postcondition
}
```
Ensuring Sanity

```cpp
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
}
```

`sane()` is the invariant of a Time object:

- holds *before* every public method
- holds *after* every public method
void Time::set_hour(int h)
{
    assert (sane());
    ...
    assert (sane());
}

bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
    (0 <= minutes() && minutes() <= 59) &&
    (0 <= seconds() && seconds() <= 60);
}

same for set_minute(), set_seconds(), etc.
Locating Infections

• Precondition failure = infection before method
• Postcondition failure = infection within method
• All assertions pass = no infection

```c++
void Time::set_hour(int h)
{
    assert (sane()); // precondition
    ...
    assert (sane()); // postcondition
}
```
Complex Invariants

class RedBlackTree {
    ...
    boolean sane() {
        assert (rootHasNoParent());
        assert (rootIsBlack());
        assert (redNodesHaveOnlyBlackChildren());
        assert (equalNumberOfBlackNodesOnSubtrees());
        assert (treeIsAcyclic());
        assert (parentsAreConsistent());

        return true;
    }
}
public aspect RedBlackTreeSanity {
    pointcut modify():
      call(void RedBlackTree.add*(..)) ||
      call(void RedBlackTree.del*(..));

    before(): modify() {
      assert (sane());
    }

    after(): modify() {
      assert (sane());
    }
}
Invariants in GDB

(gdb) break 'Time::set_hour(int)' if !sane()
Breakpoint 3 at 0x2dcf: file Time.C, line 45.
(gdb) _
Asserting Correctness
def divide(dividend, divisor):
    # Actual computation goes here
    ...
    assert quotient * divisor + remainder == dividend
    return (quotient, remainder)
void Time::set_hour(int h)
{
    // Actual code goes here

    assert (hour() == h);  // postcondition
}
void Sequence::sort()
{
    // Actual code goes here
    assert (is_sorted());
}
Postconditions

```cpp
void Container::insert(Item x)
{
    // Actual code goes here
    assert (has(x));
}
```

a helper function that is also a useful public method
Postconditions

```c++
void Heap::merge(Heap another_heap)
{
    assert (sane());
    assert (another_heap.sane());

    // Actual code goes here

    assert (sane());
}
```

Invariants are always part of pre- and postconditions
void Time::set_hour(int h)
{
    int old_minutes = minutes();
    int old_seconds = seconds();
    assert (sane());

    // Actual code goes here

    assert (sane());
    assert (hour() == h);
    assert (minutes() == old_minutes &&
            seconds() == old_seconds);
}
Contracts

set_hour (h: INTEGER) is
    -- Set the hour from `h'
require
    sane_h: 0 <= h and h <= 23
ensure
    hour_set: hour = h
    minute_unchanged: minutes = old minutes
    second_unchanged: seconds = old seconds

This contract specifies interface properties
Z Invariant

Date

hours, minutes, seconds : ℕ

0 \leq hours \leq 23
0 \leq minutes \leq 59
0 \leq seconds \leq 59
**Z Conditions**

\[
\begin{align*}
set\_hour & \quad \Delta Date \\
\h? : \mathbb{N} & \quad \text{hours} = h? \\
0 \leq h? \leq 23 & \quad \text{minutes} = \text{minutes}' \\
\end{align*}
\]

\[
\begin{align*}
\text{hours}' = h? & \quad \text{seconds} = \text{seconds}'
\end{align*}
\]
Spec vs Code

Contracts

set_hour (h: INTEGER) is
    -- Set the hour from `h'
    require
        sane_h: 0 <= h and h <= 23
    ensure
        hour_set: hour = h
        minute_unchanged: minutes = old minutes
        second_unchanged: seconds = old seconds

This contract specifies interface properties

Z Conditions

Integrated spec limited to language

Separate spec can express anything
Translated into run-time assertions

```c
/*@ requires 0 <= h && h <= 23
  @ ensures  hours() == h &&
            minutes() == \old(minutes()) &&
            seconds() == \old(seconds())
  @*/
void Time::set_hour(int h) ...
```
What does this specify?

```java
/*@ requires x >= 0.0;
@ ensures JMLDouble
@   .approximatelyEqualTo
@   (x, \result * \result, eps);
@*/
```
public class Purse {
    final int MAX_BALANCE;
    int balance;
    //@ invariant  0 <= balance && balance <= MAX_BALANCE;

    byte[] pin;
    //@ invariant  pin != null && pin.length == 4 &&
    @
   forall int i; 0 <= i && i < 4;
    @
    0 <= byte[i] && byte[i] <= 9)
    @*/

    //@ requires  amount >= 0;
    @ assignable balance;
    @ ensures  balance == \old(balance) - amount &&
    @
    \result == balance;
    @ signals (PurseException) balance == \old(balance);
    @*/

    int debit(int amount) throws PurseException { ... }
}
More use of JML

• Documentation
• Unit testing with JMLUnit
• Invariant generation with DAIKON
• Static checking with ESC/Java
• Verification with theorem provers
Relative Debugging

Rather than checking a spec, we can also compare against a reference run:

- The environment has changed—e.g. ports or new interpreters
- The code has changed
- The program has been reimplemented
Relative Assertions

• We compare two program runs

• A *relative assertion* compares variable values across the two runs:

```plaintext
assert \n   p1::perimeter@polygon.java:65 == \n   p0::perimeter@polygon.java:65
```

• Specifies when and what to compare
private void calculator() {
    numberOfSize = pr1.n; // number of sides
    ray = pr1.ray; // ray
    centerAngle = pr1.angle; // center angle
    size = pr1.size(ray, centerAngle / 2); // side
    height = pr1.height(ray, centerAngle / 2); // apotheme
    perimeter = size * numberOfSize; // perimeter
    area = perimeter * height / 2; // area
Concepts

 assertions catch infections before they propagate too far
 assertions check preconditions, postconditions and invariants
 assertions can serve as specifications
 a program can serve as reference to be compared against