Asserting Expectations
Andreas Zeller

Your Submissions
• Program must behave exactly as specified (i.e., input, output, flags, etc.)
• Program must use recent Python 2 version (i.e., Python 2.6 – installed in CIP pools)
• If we have to fix your programs, you will lose points

Oral Exams
• Some time after the course (dates set up individually, in two clusters)
During execution, the state becomes infected.

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

| what to observe | when to observe | what to expect |

Basic Assertions

```c
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

#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

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
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);
}

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

Ensuring Sanity

sane() is the invariant of a Time object:

- holds before every public method
- holds after every public method
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
}
```

Locating Infections

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

```cpp
void Time::set_hour(int h)
{
    assert (sane());  // precondition
    ...
    assert (sane());  // postcondition
}
```

Complex Invariants

[Red-black tree diagram]

Complex Invariants

```java
class RedBlackTree {
    boolean sane() {
        assert (rootHasNoParent());
        assert (rootIsBlack());
        assert (redNodesHaveOnlyBlackChildren());
        assert (equalNumberOfBlackNodesOnSubtrees());
        assert (treeIsAcyclic());
        assert (parentsAreConsistent());
        return true;
    }
}
```

Deletion

```java
void delete_one_child(struct node *p) {
    /* Precondition: p has at most one non-null child. */
    struct node *child = is_leaf(p->right) ? p->left : p->right;
    replace_node(p, child);
    if (p->color == BLACK) {
        if (child->color == RED) {
            child->color = BLACK;
        } else {
            /* delete_case1(child): */
            struct node *n = child;
            /* this loop performs tail recursion on delete_case1(n) */
            for (;;) {
                /* delete_case1(n): */
                if (n->parent != NULL) {
                    /* delete_case2(n): */
                    struct node *s;
                    s = sibling(n);
                    if ((n->parent->color == BLACK) &&
                        (s->color == BLACK) &&
                        (s->left->color == BLACK) &&
                        (s->right->color == BLACK)) {
                        s->color = RED;
                        /* delete_case1(n->parent): */
                        n = n->parent;
                        continue;
                    } else {
                        /* delete_case4(n): */
                        //s = sibling(n);//not needed
                        if ((n->parent->color == RED) &&
                            (s->color == BLACK) &&
                            (s->left->color == BLACK) &&
                            (s->right->color == BLACK)) {
                            s->color = RED;
                            n->parent->color = BLACK;
                            break;
                        } else {
                            /* delete_case5(n): */
                            //s = sibling(n);//not needed
                            if ((n == n->parent->left) &&
                                (s->color == BLACK) &&
                                (s->left->color == RED) &&
                                (s->right->color == BLACK)) {
                            s->color = RED;
                            s->left->color = BLACK;
                            rotate_right(s);
                        } else if ((n == n->parent->right) &&
                                (s->color == BLACK) &&
                                (s->right->color == RED) &&
                                (s->left->color == BLACK)) {
                            s->color = RED;
                            s->right->color = BLACK;
                            rotate_left(s);
                        } /* delete_case6(n): */
                        s = sibling(n);
                        s->color = n->parent->color;
                        n->parent->color = BLACK;
                        if (n == n->parent->left) {
                            /* Here, s->right->color == RED. */
                            s->right->color = BLACK;
                            rotate_left(n->parent);
                        } else {
                            /* Here, s->left->color == RED. */
                            s->left->color = BLACK;
                            rotate_right(n->parent);
                        }
                        break;
                    } /* terminate tail recursion */
                } else {
                    /* delete_case3(n): */
                    s = sibling(n);
                    if ((n->parent->color == BLACK) &&
                        (s->color == BLACK) &&
                        (s->left->color == BLACK) &&
                        (s->right->color == BLACK)) {
                    s->color = RED;
                    /* delete_case1(n->parent): */
                    n = n->parent;
                    continue;
                } else {
                    /* delete_case5(n): */
                    //s = sibling(n);//not needed
                    if ((n == n->parent->left) &&
                        (s->color == BLACK) &&
                        (s->left->color == RED) &&
                        (s->right->color == BLACK)) {
                    s->color = RED;
                    s->left->color = BLACK;
                    rotate_right(s);
                } else if ((n == n->parent->right) &&
                                (s->color == BLACK) &&
                                (s->right->color == RED) &&
                                (s->left->color == BLACK)) {
                            s->color = RED;
                            s->right->color = BLACK;
                            rotate_left(s);
                        } /* delete_case6(n): */
                        s = sibling(n);
                        s->color = n->parent->color;
                        n->parent->color = BLACK;
                        if (n == n->parent->left) {
                            /* Here, s->right->color == RED. */
                            s->right->color = BLACK;
                            rotate_left(n->parent);
                        } else {
                            /* Here, s->left->color == RED. */
                            s->left->color = BLACK;
                            rotate_right(n->parent);
                        }
                        break;
                    } /* terminate tail recursion */
                }
            } /* tail recursion loop */
        } /* delete_case3(n): */
    } /* delete_case2(n): */
    else {
        /* delete_case1(child): */
        struct node *n = child;
        /* this loop performs tail recursion on delete_case1(n) */
        for (;;) {
            /* delete_case1(n): */
            if (n->parent != NULL) {
                /* delete_case2(n): */
                struct node *s;
                s = sibling(n);
                if ((n->parent->color == BLACK) &&
                    (s->color == BLACK) &&
                    (s->left->color == BLACK) &&
                    (s->right->color == BLACK)) {
                s->color = RED;
                /* delete_case1(n->parent): */
                n = n->parent;
                continue;
            } else {
                /* delete_case3(n): */
                s = sibling(n);
                if ((n->parent->color == BLACK) &&
                    (s->color == BLACK) &&
                    (s->left->color == BLACK) &&
                    (s->right->color == BLACK)) {
                s->color = RED;
                n->parent->color = BLACK;
                break;
            } else {
                /* delete_case4(n): */
                //s = sibling(n);//not needed
                if ((n->parent->color == RED) &&
                    (s->color == BLACK) &&
                    (s->left->color == BLACK) &&
                    (s->right->color == BLACK)) {
                s->color = RED;
                s->left->color = BLACK;
                rotate_right(s);
            } else if ((n->parent->color == RED) &&
                                    (s->color == BLACK) &&
                                    (s->left->color == BLACK) &&
                                    (s->right->color == BLACK)) {
                        s->color = RED;
                        s->right->color = BLACK;
                        rotate_left(s);
                    } /* delete_case6(n): */
                    s = sibling(n);
                    s->color = n->parent->color;
                    n->parent->color = BLACK;
                    if (n == n->parent->left) {
                        /* Here, s->right->color == RED. */
                        s->right->color = BLACK;
                        rotate_left(n->parent);
                    } else {
                        /* Here, s->left->color == RED. */
                        s->left->color = BLACK;
                        rotate_right(n->parent);
                    }
                    break;
                } /* terminate tail recursion */
            } /* tail recursion loop */
        } /* this loop performs tail recursion on delete_case1(n) */
    } /* delete_case3(n): */
}
```


Invariants as Aspects

```java
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)
Postconditions

```cpp
void Time::set_hour(int h)
{
    // Actual code goes here
    assert (hour() == h);   // postcondition
}
```

Postconditions

```cpp
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

- helper function
Postconditions

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

Checking Earlier State

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

<table>
<thead>
<tr>
<th>Date</th>
<th>hours, minutes, seconds : ( \mathbb{N} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( 0 \leq \text{hours} \leq 23 )</td>
</tr>
<tr>
<td></td>
<td>( 0 \leq \text{minutes} \leq 59 )</td>
</tr>
<tr>
<td></td>
<td>( 0 \leq \text{seconds} \leq 59 )</td>
</tr>
</tbody>
</table>

seconds can be 60! Make a point about errors in specs

Z Conditions

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

Spec vs Code

Contracts

\[
\begin{align*}
\text{set}_\text{hour} (h : \text{INTEGER}) & : \Delta \text{Name} \\
& : \ h \geq 0 \text{ and } h = 23 \\
\text{require} & : \ h \geq 0 \text{ and } h \leq 23 \\
\text{ensure} & : \ h \text{ unchanged: } \text{minutes} = \text{old minutes} \\
& : \ h \text{ unchanged: } \text{seconds} = \text{old seconds}
\end{align*}
\]

This contract specifies interface properties

Integrated spec limited to language

Separate spec can express anything

Separate spec can express anything
/*@ requires 0 <= h && h <= 23 @*/
void Time::set_hour(int h) … 

Translated into run-time assertions

/*@ requires x >= 0.0; @*/
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 <= pin[i] && pin[i] <= 9)
  @*/
  /*@ requires amount >= 0; @*/
  int debit(int amount) throws PurseException { … }

Developed by Gary Leavens et al, now a cooperative effort of dozens of researchers
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:
  
  ```
  assert p1::perimeter@polygon.java:65 == p0::perimeter@polygon.java:65
  ```
- Specifies when and what to compare
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