Search in Time

- During execution, the state becomes infected.
- Basic idea: Observe a transition from sane to infected.

Manual Observation
Automated Observation

<table>
<thead>
<tr>
<th>what</th>
<th>when</th>
<th>what</th>
</tr>
</thead>
<tbody>
<tr>
<td>to observe</td>
<td>to observe</td>
<td>to expect</td>
</tr>
</tbody>
</table>

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

```c
#ifndef NDEBUG
#define assert(ex) ((ex) ? 1 : (cerr << __FILE__ << "": 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

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

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
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
    assert (sane());
    same for set_minute(),
    set_seconds(), etc.
}
```

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
    ... // postcondition
    assert (sane());
}
```

Complex Invariants

Complex Invariants

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

Deletion


Invariants as Aspects

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());
}

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

a helper function that is also a useful public method
Postconditions

```cpp
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

```cpp
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

```plaintext
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

\[
\begin{align*}
\text{Date} & \quad \text{hours, minutes, seconds} : \mathbb{N} \\
0 & \leq \text{hours} \leq 23 \\
0 & \leq \text{minutes} \leq 59 \\
0 & \leq \text{seconds} \leq 59 \\
\end{align*}
\]

seconds can be 60!

Make a point about errors in specs

Z Conditions

\[
\begin{align*}
\text{set} \ hour \ (h: \text{INTEGER}) \ & \text{is} \\
\text{require} & \quad \text{sane} \ h: 0 \leq h \text{ and } h \leq 23 \\
\text{ensure} & \quad \text{hour} \ _\text{set}: \ 	ext{hour} = h \\
& \quad \text{minute} \ _\text{unchanged}: \ 	ext{minutes} = \text{old} \ 	ext{minutes} \\
& \quad \text{second} \ _\text{unchanged}: \ 	ext{seconds} = \text{old} \ 	ext{seconds} \\
\end{align*}
\]

Spec vs Code

Contracts

\begin{verbatim}
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
\end{verbatim}

Separate spec can express anything

Integrated spec limited to language

Integrated spec limited to language

Separate spec can express anything

Separate spec can express anything
Translated into run-time assertions

What does this specify?

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 == \n  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

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/1.0 or send a letter to Creative Commons, 559 Abbott Way, Stanford, California 94305, USA.