The following case study illustrates the application of black-box test case design techniques to a relatively small but “real-world” specification for a standard mathematical library function: `pow()`. (Slightly simplified versions of actual MAN pages are used.)
Assumptions

- We will assume that the function in question is currently under development.
- You are part of the test design team in an independent, system-level testing group.
- In addition to draft user MAN pages, relevant mathematical reference material and software specification documents are available.
Mathematical Library

NAME

pow - power function

SYNOPSIS

pow(x,y);

DESCRIPTION

The pow() function computes the value of x raised to the power y, \( x^{**}y \). If x is negative, y must be an integer value.
Mathematical Library

RETURN VALUES

Upon successful completion, pow() returns the value of x raised to the power y.

If x is 0 and y is 0, 1 is returned.

If x or y is non-numeric, NaN is returned.

If x is 0 and y is negative, +HUGE_VAL is returned.

If the correct value would cause overflow, +HUGE_VAL or -HUGE_VAL is returned.

If the correct value would cause underflow, +TINY_VAL or -TINY_VAL is returned.

For exceptional cases, matherr is called and returns a value (via errno) describing the type of exception that has occurred.
An application wishing to check for error situations should set `errno` to 0 before calling `pow()`. If `errno` is non-zero on return, or if the return value of `pow()` is `NaN`, an error has occurred.
Issues

• Are the requirements reflected in the draft MAN page for pow complete? consistent? unambiguous? verifiable?

• Should `matherr` and the value of `errno` be taken into account when designing tests for pow?

• Is it clear what pow does in “error cases”? (E.g., is matherr called when x or y is not a number? Is 0**0 an “error case”?)
Mathematical Library

NAME

matherr - math library exception-handling function

SYNOPSIS

matherr;

DESCRIPTION

Certain math functions call matherr when exceptions are detected. matherr stores an integer in errno describing the type of exception that has occurred, from the following list of constants:

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERFLOW</td>
<td>overflow range exception</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>underflow range exception</td>
</tr>
</tbody>
</table>
Excerpts from the matherr MAN page (cont’d)

Mathematical Library

STANDARD CONFORMANCE

The following table summarizes the pow exceptions detected, the values returned, and the constants stored in `errno` by matherr.

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th>OVERFLOW</th>
<th>UNDERFLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>usual cases</td>
<td>+/-HUGE_VAL</td>
<td>+/-TINY_VAL</td>
</tr>
<tr>
<td>0**0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(x&lt;0)**(not int)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0**(y&lt;0)</td>
<td>+HUGE_VAL</td>
<td></td>
</tr>
</tbody>
</table>

The constants **HUGE_VAL** and **TINY_VAL** represent the maximum and minimum single-precision floating-point numbers for the given environment, respectively.
Notes from the software specification documents...

- Overflow occurs if and only if the actual value of $x^y$ is larger than +/-HUGE_VAL. Similarly, underflow occurs if and only if the actual non-zero value of $x^y$ is smaller than +/-TINY_VAL. (Note that +HUGE_VAL is distinct from -HUGE_VAL; similarly for +TINY_VAL and -TINY_VAL.)

- Like DOMAIN, OVERFLOW, and UNDERFLOW, “NaN” (Not a Number) is a system-defined constant.
Black-box design strategies

• Which black-box design strategies are appropriate?
  – Equivalence Partitioning?
  – Cause-Effect Analysis?
  – Boundary Value Analysis?
  – Intuition and Experience?

• In what order should those that are appropriate be applied?
1. Develop a simple **Cause-Effect Model** based on the MAN pages and other documentation available.

   - Identify Causes and Effects.
   - Describe relationships and constraints deducible from specification via Cause-Effect graphs that could serve as a basis for test case design.
A possible scheme... (cont’d)

2. Identify specific Boundary Values that should/could be covered using additional test cases.

3. Identify other specific conditions or usage scenarios that may be worthwhile to cover by applying Intuition & Experience.
Developing a Cause-Effect Model

1. What **Effects** should be modeled for pow()?
   - Should *matherr* be considered?
   - Should they be mutually exclusive or not?

2. What **Causes** should be modeled?
   - Should they be based on pairwise (x,y) value classes, or independent x and y value classes? (E.g., should "x=0 & y<0" be a single Cause or should "x=0" and "y<0" be two separate Causes?)
Some Non-Mutually Exclusive Effects…

21. pow returns x**y
22. pow returns 0
23. pow returns 1
24. pow returns NaN
25. pow returns +/- Huge_Val
26. pow returns +/- Tiny_Val
27. matherr returns OVERFLOW
28. matherr returns UNDERFLOW
29. matherr returns DOMAIN
And Some Corresponding Causes

1. $x=0$
2. $x<0$
3. $x>0$
4. $y=0$
5. $y<0$
6. $y>0$
7. $y$ is an integer
8. $x$ not a number (NaN)
9. $y$ not a number (NaN)
10. $x**y > \text{Huge}_\text{Val}$
11. $x**y < -\text{Huge}_\text{Val}$
12. $0 < x**y < \text{Tiny}_\text{Val}$
13. $0 > x**y > -\text{Tiny}_\text{Val}$
Alternative Approach: Mutually Exclusive Effects

Values RETURNED by pow() and SET (via errno) by matherr:

21. pow: NaN, matherr not called

---

22. pow: +Huge_Val, matherr: OVERFLOW
23. pow: -Huge_Val, matherr: OVERFLOW
24. pow: +Tiny_Val, matherr: UNDERFLOW
25. pow: -Tiny_Val, matherr: UNDERFLOW
26. pow: 1, matherr: DOMAIN
27. pow: 0, matherr: DOMAIN
28. pow: +Huge_Val, matherr: DOMAIN

---

31. pow: returns \( x^{y} \) (???)}, matherr: not called
The draft MAN page primarily describes exceptional/error behaviors of pow(). What non-error Effect(s) should be modeled?

An implicit part of the specification for pow() includes the mathematical definition of exponentiation. Consider, for example, the following excerpts from a popular math dictionary:
Excerpts from a mathematical definition of pow()

"If the exponent \( y \) is a positive integer, then \( x^{**}y \) MEANS \( x \) if \( y=1 \), and it MEANS the product of \( y \) factors each equal to \( x \) if \( y>1 \)."

In other words, when \( y \) is an integer \( \geq 1 \), the function associated with \( x^{**}y \) is DEFINED to be

\[
\prod_{i=1}^{y} x
\]
Excerpts from a mathematical definition of pow() (cont’d)

Similarly, the dictionary states:

"If x is a nonzero number, the value of x**0 is *DEFINED* to be 1." … "A negative exponent indicates that…the quantity is to be reciprocated. …If the exponent is a fraction p/q, then x**(p/q) is *DEFINED* as (x**(1/q))**p, where x**(1/q) is the positive qth root of x if x is positive…"
Non-error Effects (cont’d)

• Thus, \( x^{**}y \) really MEANS different functions depending on what region (point, line, etc.) in the \( x,y \) plane we are talking about.

• Clearly, these differences could be reflected in separate non-error Effects. For example, you might define an Effect to be:

\[
\text{"pow returns } \prod_{i=1}^{y} x \text{ and matherr is not called"}
\]
Non-error Effects (cont’d)

• The Causes for this Effect could be modeled as "y is a whole number ≥1" (a single Cause) AND'ed with an intermediate node representing "NOT any of the error conditions".

• We can therefore expand our single non-error Effect to:

  31. pow: the product of y factors, each equal to x, matherr: not called
  32. pow: 1 by defn. of (x<>0)**0, matherr: not called
  33. pow: the reciprocal of x**(-y), matherr: not called
  34. pow: a positive root of x is computed, matherr: not called
Corresponding Causes

pow() is called with arguments x,y such that...

1. x not a number (NaN)
2. y not a number (NaN)
---
3. $x^{**}y > \text{Huge\_Val}$ - the mathematical value of input $x$ raised to the power of input $y$ exceeds the constant $\text{+Huge\_Val}$
4. $x^{**}y < -\text{Huge\_Val}$ - (similar to above defn.)
5. $0 < x^{**}y < \text{Tiny\_Val}$ - the mathematical value of input $x$ raised to the power of input $y$ is greater than 0 and less than $\text{Tiny\_Val}$
6. $0 > x^{**}y > -\text{Tiny\_Val}$ - (similar to above defn.)
Corresponding Causes (cont’d)

7. $x=0, \ y=0$
8. $x<0, \ y \text{ not an integer}$
9. $x=0, \ y<0$

---

10. $y \text{ an integer } \geq 1$
11. $x<>0, \ y=0$
12. $y<0$
13. $x\geq0, \ y \text{ not an integer}$
Cause-Effect Graphs

Error Situations:

1. $x$ NaN
2. $y$ NaN
3. $x^y > \text{Huge}_\text{Val}$
4. $x=0$, $y<0$

- $x$ NaN: pow: NaN, matherr not called
- $y$ NaN: pow: NaN, matherr not called
- $x^y > \text{Huge}_\text{Val}$: pow: +Huge_Val, matherr: OVERFLOW
- $x=0$, $y<0$: pow: +Huge_Val, matherr: DOMAIN
Cause-Effect Graphs (cont’d)

Non-Error Situations (for use with Effects 31-34):

1 \rightarrow \text{NaN} \rightarrow \text{NES}

\ldots

9 \rightarrow \text{NES}
Cause-Effect Graphs (cont’d)
Identifying Boundary Values

- **x, y input** boundary points/lines/regions:
  - \( x=0, y=0 \) (pow: 1, matherr: domain)
  - \( x<0, y \text{ not an integer} \) (pow: 0, matherr: domain)
  - \( x=0, y<0 \) (pow: +HUGE_VAL, matherr: domain)
  - \( y=1 \) (pow: x)
  - \( y \text{ an integer } > 1 \) (pow: \( \prod_{i=1}^{y} x \) )
  - \( x\neq0, y=0 \) (pow: 1)
  - \( x\neq0, y<0 \) (pow: reciprocal of \( x^{**(-y)} \) )
Identifying Boundary Values (cont’d)

- `pow()` **output** boundaries:
  - +/- HUGE_VAL
  - +/- TINY_VAL
  - +/- 0

![Diagram](image)
Exploring Boundaries

x=0, y=0
$x \neq 0, \ y = 0$
Exploring Boundaries (cont’d)

$x \neq 0, y = 0$
Exploring Boundaries (cont’d)

\[ x^y = +\text{HUGE}_\text{VAL} \]
Exploring Boundaries (cont’d)

\[ x^y = +\text{HUGE\_VAL} \]
Intuition and Experience

• Various input TYPE errors (non-numeric chars, blanks, pointers, functions, etc.), including a mix of: no arguments, 1 argument, > 2 arguments

• Various combinations of system-defined constant inputs:
  - +/- HUGE_VAL, +/- TINY_VAL
  - +/- INF, +/- 0, NaN

• All documented examples (including invalid or singular inputs)

• System rounding mode variations, etc.
Case Study: Black-Box Testing

Software Testing and Verification

Lecture 6.1

Prepared by
Stephen M. Thebaut, Ph.D.

University of Florida