



CubeSat Flight Software Workshop

Reducing Risk

Gerard J. Holzmann





June 4, 2019



Jet Propulsion Laboratory
California Institute of Technology

reducing software risk – part 1: awareness of bounds

- we would like to believe that we can use math to reason about code, and that it's a simple matter of doing the right conversions:

- abstract domain  concrete domain
- natural numbers  integers
- real numbers  floating point numbers
- induction  recursion

but: everything on the *right* has limited precision, while everything on the *left* does not

converting from math to code requires careful consideration of:

- resource limits (time/memory)
- resource sharing (concurrency)
- computational complexity

part 1: awareness of bounds

- be aware of bounds when dealing with:
 - numbers (data: precision, rounding)
 - cycles (time: loops, complexity)
 - bytes (memory: heaps and stacks are finite)

bounds on data: computing factorials

a simple recurrence relation

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases} \quad \forall n \in \mathbb{N}.$$

```
#include <stdio.h>
#include <assert.h>

int fact(int n) { return (n == 0) ? 1 : n*fact(n-1); }

int
main(int argc, char *argv[])
{
    int n;

    if (argc != 2)
    {
        printf("usage: fact N\n");
        exit(1);
    }

    n = atoi(argv[1]);
    assert(n >= 0);

    printf("%d! = %d\n", n, fact(n));
    exit(0);
}
```

```
$ for i in `seq 13`
do ./fact $i
done
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
```

```
1932953504 / 479001600 =
4.03538005718561
13*12*11*10*9*8*7*6*5*4*3*2*1 =
6227020800
6227020800 - 2^32
1932053504
```

bounds on loops: the microsoft zune 30 GB

stopped working midnight december 31, 2008



zune 30 faq

My Zune 30 is frozen. What should I do?

Follow these steps:

1. Disconnect your Zune from USB and AC power sources.
2. Because the player is frozen, its battery will drain—this is good. Wait until the battery is empty and the screen goes black. If the battery was fully charged, this might take a couple of hours.
3. Wait until after noon GMT on January 1, 2009 (that's 7 a.m. Eastern or 4 a.m. Pacific time).
4. Connect your Zune to either a USB port on the back or your computer or to AC power using the Zune AC Adapter and let it charge.

Once the battery has sufficient power, the player should start normally. No other action is required—you can go back to using your

3. Wait until after noon GMT on January 1, 2009 (that's 7 a.m. Eastern or 4 a.m. Pacific time).

My Zune 30 has been working fine today. Should I be worried?

Nope, your Zune is fine and will continue to work as long as you do not connect it to your computer before noon GMT on January 1, 2009 (7 a.m. Eastern or 4 a.m. Pacific time).

Note: If you connect your player to a computer before noon GMT on January 1, 2009, you'll experience the freeze mentioned above—even if that computer does not have the Zune software installed. If this happens, follow the above steps.

What if I have rights-managed (DRM) content on my Zune?

Most likely, rights-managed content will not be affected by this issue. However, it's a good idea to sync your Zune with your computer once the freeze has been resolved, just to make sure your usage rights are up to date.

What if I took advice from the forums and reset my Zune by disconnecting the battery?

This is a bad idea and we do not recommend opening your Zune by yourself (for one thing, doing so will void your warranty). However, if you've already opened it, do one of the following:

- Wait 24 hours from the time that you reset the Zune and then sync with your computer to refresh the usage rights; or
- Delete the player's content using the Zune software (go to Settings, Device, Sync Options, Erase All Content), then re-sync it from your collection.



source:
public Microsoft FAQ webpage
for the Zune 30, Dec. 2008

bounds on loops: the zune code

input : days elapsed since Jan 1, 1980

output: year + day of year

```
year = 1980;
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    } else
    {
        days -= 365;
        year += 1;
    }
}
```

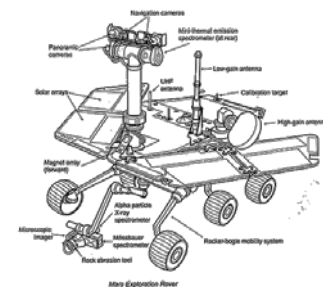
December 31, 2008 was the 366th day of the year.

(2008 was a leap year: a multiple of 4, but not of 100 or 400).

-> the zune got stuck in an infinite loop

bounds on memory: **finite resources**

example: MER Sol-18 (2004)



Rule1: when a process runs out of memory, it is suspended
[the system is designed in such a way that this cannot happen]

Rule2: if a “cannot happen” condition is seen, the system reboots

Q: so, what happens when a “cannot happen” condition occurs
during a reboot?

A: that cannot happen

Jan. 21, 2004 – 18 days after the first MER Rover landed on Mars

The “cannot happen” scenario:

→ the flash-file system accumulated more data than anticipated
file system meta-data was reconstructed in core when the file
system is mounted during a reboot
more heap memory needed than available → suspension → reboot

reducing software risk – part 2: defensive coding

“a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage patterns” (*wikipedia*)

- mitigating the effects of Murphy's Law.

part 2: defensive coding example

```
void  
vulnerable (char *input)  
{ char str[1024];  
  
  strcpy(str, input);  
  ...  
}
```

```
int  
defensive( const char *input )  
{ char str[1024];  
  
  if (input == NULL)  
  { return ERROR;  
  }  
  
  strncpy(str, input, sizeof(str));  
  str[sizeof(str)-1] = '\0'; // null terminate  
  
  ...  
  return SUCCESS;  
}
```

*let the compiler help you
catch glitches*

*if this is an expected case
if not: use “**assert**(input != NULL)”*

*never use strcpy()
always strncpy()*

make fewer assumptions
in this case: about the validity of parameters
(now, and in future revisions of the code)

use assertions
to indicate assumed “cannot happen” cases.
because when they do happen, you want to know

defensive coding: bounded loops

- in mission critical code, loops must be *provably bounded*
 - in such a way that a static analysis tool can *verify* this
 - never modify a loop index inside the body of the loop
 - if there is no clear bound, create one (e.g., INT_MAX):

```
int cnt = 0;
for (ptr = head; ptr != NULL; ptr = ptr->nxt, cnt++)
{
    assert(cnt < INT_MAX);
    ...
}
```

defensive coding: assertion density and defect density

(a Microsoft study MSR-TR-2006-54, pub: Proc. ISSRE 2006, pp. 204-212)

Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation

Gunnar Kudrjavets¹, Nachiappan Nagappan², Thomas Ball²

¹Microsoft Corporation, Redmond, WA 98052

²Microsoft Research, Redmond, WA 98052

{gunmarku, nachin, tball}@microsoft.com

Abstract

The use of assertions in software development is thought to help produce quality software. Unfortunately, there is scant empirical evidence in commercial software systems for this argument to date. This paper presents an empirical case study of two commercial software components at Microsoft Corporation. The developers of these components systematically employed assertions, which allowed us to investigate the relationship between software assertions and code quality. We also compare the efficacy of assertions against that of popular bug finding techniques like source code static analysis tools. We observe from our case study that with an increase in the assertion density in a file there is a statistically significant decrease in fault density. Further, the usage of software assertions in these components found a large percentage of the faults in the bug database.

Keywords: Assertions, Faults, Bug database, Source control systems, Correlations.

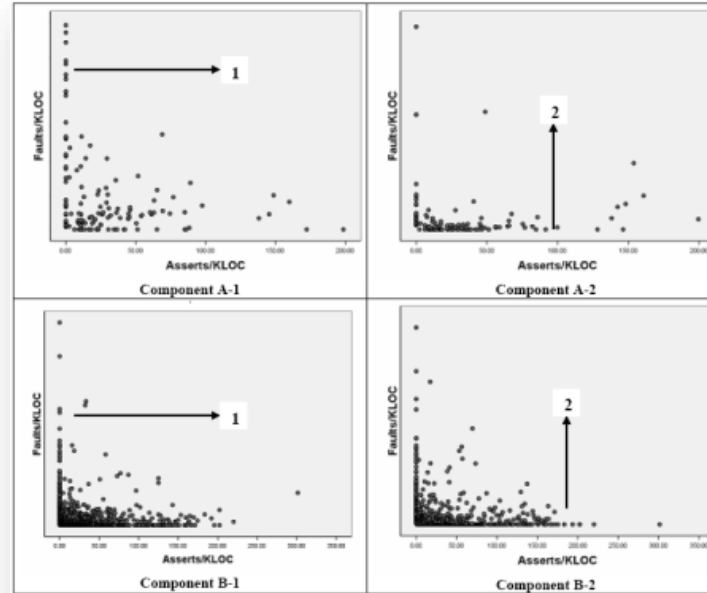


Figure 3: Scatter plots between assertion density and fault density for components A, B

“with an increase in assertion density there is a statistically significant decrease in fault density”

defensive coding: **coding standards**

follow a machine-checkable, risk-based, standard

1. Restrict to simple control flow constructs
2. Do not use recursion and give all loops a fixed upper-bound
3. Do not use dynamic memory allocation after initialization
4. Limit functions to no more than ~60 lines of text
5. Target an average assertion density of 2% per module
6. Declare data objects at the smallest possible level of scope
7. Check the return value of non-void functions; check the validity of parameters
8. Limit the use of the preprocessor to file inclusion and simple macros
9. Limit the use of pointers. Use no more than 2 level of dereferencing
10. Compile with all warnings enabled, and use source code analyzers



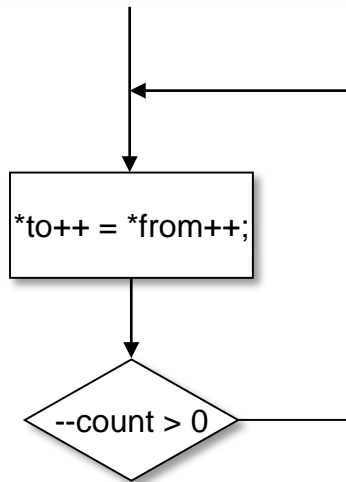
IEEE Computer 39(6) 95-97 (2006)

<http://spinroot.com/p10/>

defensive coding

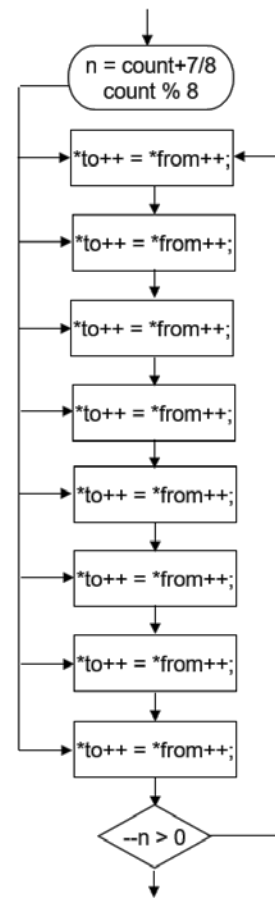
use simple control-flow constructs...

```
do {  
    *to++ = *from++;  
} while(--count > 0);
```



```
/* assuming count > 0 */  
int n = (count + 7) / 8;  
  
switch (count%8)  
{  
    case 0: do { *to++ = *from++;  
    case 7:      *to++ = *from++;  
    case 6:      *to++ = *from++;  
    case 5:      *to++ = *from++;  
    case 4:      *to++ = *from++;  
    case 3:      *to++ = *from++;  
    case 2:      *to++ = *from++;  
    case 1:      *to++ = *from++;  
                } while (--n > 0);  
}
```

"Duff's device," 1983



defensive coding: testable code

example: make very limited use of the C preprocessor

Q1: will this code trigger
a compilation error (and if so, where)?

```
#define A

int
main(void)
{
    #ifndef A
        printf("hello world\n");
    #else1
        goto *main();
    #endif
    return 0;
}
```

```
$ gcc -Wall -pedantic gobble.c
$ ./a.out
$
```

Q2: how many different ways can this
code be compiled (i.e., how many
ways would it need to be tested)?

```
#if (a>0)
....
    #ifdef X
        ....
            #ifndef Y
                ...
                    #if b
                        ...
                    #endif
                ...
            #endif
        ...
    #endif
...
#endif
```

A: 16 different ways (2^4)

defensive coding: **secure language compliance**

avoid unspecified, undefined, or implementation defined code

- **Unspecified**

The compiler has to make a choice from a finite set of alternatives, but that choice is not in general predictable by the programmer.

Example: *the order in which sub-expressions are evaluated in a C expression.*

- **Implementation defined**

The compiler has to make a choice, and the choice required to be documented and available to the programmer,

Example: *the range of C variables of type short, int, and long.*

- **Undefined**

The language definition gives no indication of what behavior can be expected from a program – it may be some form of catastrophic failure (a ‘crash’) or continued execution with arbitrary data.

Example: *dereferencing a null pointer in C.*

defensive code: C is more powerful than you think

and therefore more risky

```
#include <stdio.h>

void *f(void)
{
a:
    printf("here twice\n");
    return &a;
}

int main(int ac, char **av)
{
    goto *f();
    printf("never here\n");
    return 0;
}
```

the C language standard says:
“a label is an *identifier*”

```
$ cc -o goto goto.c
$ cc -Wall -o goto goto.c
$ ./goto
here twice
here twice
$
```


summary

1. be aware of bounds
 - math and code are very different
 - be aware of bounds on:
 - data (numbers), loops (time), and, memory (resources)
2. use defensive coding strategies
 - don't assume: assert
 - assertions flag the “cannot happen” cases
 - assertions are not for error handling (expected cases)
3. follow, *and check compliance*, with a risk-based coding standard
 - always compile with *all* warnings enabled
 - use strong static analyzers on every build

static source code analyzers, quick reference

- the best commercial source code analyzers:
 - [coverity](#) (easy to install, conservative)
 - [codesonar](#) (somewhat deeper analysis, less conservative)
 - [klocwork](#) (best array bounds checking)
 - [semmle](#) (easier to extend, rule libraries)
- open source code analyzer
 - cobra, www.spinroot.com/cobra (fast, extendable, interactive)
- for open source code, some commercial analyzers can be free:
 - <https://lgtm.com> (semmle)
 - <http://github.com> (coverity)
 - www.mir-swamp.org (codesonar)

further reading

- Steve Maguire, *Writing Solid Code*, Microsoft Programming Series, 1993
- Steve McConnell, *Code Complete*, 2nd Edition, Microsoft Programming Series, 2004
- Nancy Leveson, *Safeware*, Addison-Wesley, 1995
- Charles Perrow, *Normal Accidents*, Princeton University Press, 1999
- Gerard Holzmann, *The Power of Ten – Rules for Developing Safety Critical Code*, *IEEE Computer*, June 2006, pp. 93-95, (<http://spinroot.com/p10>)



Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov