New Study of Obesity Looks for Larger Test Group

Neurosis is the inability to tolerate ambiguity.
--Sigmund Freud

CSE503: SOFTWARE ENGINEERING

PROGRAMS, BEHAVIORS, AMBIGUITY

David Notkin
Spring 2011

International Obfuscated C Code Contest http://www.ioccc.org/2004/hoyle


```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define _                                ;double
#define void                             x,x
#define case(break,default)              break[O]:default[O]:
#define switch(bool)                     ;for(;x<bool;
#define do(if,else)                      inIine(else)>int##if?
#define true                             (--void++)
#define false                            (++void--)

char*O=" <60>!?\\\n"_ doubIe[010]_ int0,int1 _ Iong=0 _ inIine(int eIse){int
O1O=!O _ l=!O;for(;O1O<010;++O1O)l+=(O1O[doubIe]*pow(eIse,O1O));return l;}int
main(int booI,char*eIse[]){int I=1,x=-*O;if(eIse){for(;I<010+1;I++)I[doubIe-1]
=booI>I?atof(I[eIse]):!O switch(*O)x++)abs(inIine(x))>Iong&&(Iong=abs(inIine(x
)));int1=Iong;main(-*O>>1,0);}else{if(booI<*O>>1){int0=int1;int1=int0-2*Iong/0
[O]switch(5[O]))putchar(x-*O?(int0>=inIine(x)&&do(1,x)do(0,true)do(0,false)
case(2,1)do(1,true)do(0,false)6[O]case(-3,6)do(0,false)6[O]-3[O]:do(1,false)
case(5,4)x?booI?0:6[O]:7[O])+*O:8[O]),x++;main(++booI,0);}}}
```

Why is this program distasteful?

Because…

- Indentation?
- Documentation?
- Behaviors?
- Structure?
- Reasoning – loops, invariants?
- Fixing it?
- Changing it?
- …

- Programs have three immediate audiences
  - The computer
  - The developers
  - The users
- Given that this program compiles and executes as intended, the computer is perfectly happy
- Under almost no conditions are the developers happy with this program
- What about the users?

Software engineering…

- …is primarily concerned with the "happiness" of the software engineering team and with the "happiness" of the users
  - The "happiness" of the computer (performance, etc.) is material, but less so
  - We will focus more overall on the software engineering team than on the users – due largely to my knowledge and interests
- The developers need to be able to – at reasonable cost, whatever that means – understand, reason about, fix, change, enhance, etc. the program

Performance: quotations

5

503 will not discuss much more about performance – it's important in many systems, but it's not our focus

- Michael Jackson
  - Rule 1: Don't do it.
  - Rule 2 (for experts only): Don't do it yet.
- Bill Wulf
  - More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.
- Don Knuth
  - We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Why is software engineering hard?

6

You go first

Validation vs. verification

7

Building the system right (verification) vs. building the right system (validation) –Barry Boehm

- Distinct objectives intertwined in non-obvious ways – the distinction itself is often poorly understood or ignored
- Changes to the system's requirements cause changes to the implementation
- Difficulties in implementation can cause (the need for) changes to the requirements

"There are two ways to write error-free programs; only the third one works." –Perlis

Dominant discipline – Stu Feldman

8

10^3 Lines of Code	Mathematics
10^4 LOC	Science
10^5 LOC	Engineering
10^6 LOC	Social Science
10^7 LOC	Politics
10^8 LOC, 10^9 LOC, …	???, ???, …

Design under constraints

9

- Software, like other engineered entities, is designed and built under constraints
- Some of the constraints are explicit and many are implicit
- Constraints are broad, ranging across customer needs, shipping deadlines, resource limitations (memory, power, money, etc.), compatibility, reward structure, organizational culture, and much more…

A consequence of varied constraints

10

- There is *no single right way to engineer software:* no best programming language, design method, software process, testing approach, team structure, etc.
- This does not imply that every approach is good under some constraints
- Nor does it suggest that there are no consistent themes across effective approaches
- But committing to a single "best approach" can be limiting

"Please don't fall into the trap of believing that I am terribly dogmatical about [the goto statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!"
–Dijkstra

"Don't get your method advice from a method enthusiast. The best advice comes from people who care more about your problem than about their solution."
–M. Jackson

Complexity

11

"Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one… In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound."
—Brooks

Complexity and people – Dijkstra

12

- "The competent programmer is fully aware of the limited size of his own skull."
- "Software is so complex that our poor head cannot cope with it at all. Therefore, we have to use all possible means and methods to try to control this complexity."

Size
50MLOC = 50 million lines of code

13

- 50 lines/page-side ⇒ 1M page-sides
- 1K page-sides/ream ⇒ 1K reams
- 2 inches/ream ⇒ 2K inches
- 2K inches = 167 feet ≈ twice the height of the Allen Center
- 5 words/LOC @ 50 wpm ⇒ 50MLOC/5M min
- 5M min = 83,333 hr = 3,472 days ≈ 10 years
- Just for typing … no fair thinking!

Design space complexity [Jackson]

14

- Designing both automobiles and bridges requires *specialized* knowledge
- Automobile design is *standardized*: the designers know virtually everything about the context in which the automobile will be used: expected passenger weights, what kind of roads will be encountered, etc.
- But bridge design is *not standardized*: the designers must understand the specific location in which the bridge will be built: the length of the span, the kind of soil, the expected traffic, etc.

Software design space

15

- Software design is widely and wildly *non-standardized* (as well as being *specialized*)
- Figuring out what the user wants and needs is hard and is almost always part of the job; for most software systems, this goes far beyond designing a bridge for a specific location
- A classic exception is some classes of compilers
  - The PQCC project at CMU (Wulf et al., 1980) led to the formation of Tartan Laboratories, which was acquired by TI (1996) primarily to construct C compilers for DSPs – in essence, this became standardized
  - Jackson suggests that "compiler engineering" (and such) might make sense, in contrast to "software engineering"

All useful programs undergo continuing change
Belady and Lehman (1976)

A significant amount of "software maintenance" makes changes for which roughly analogous changes would be considered non-routine in most other fields

- Adding floors to skyscrapers, lanes to bridges
- Accommodating new aircraft at airports
- Adding Cyrillic-based languages to European Union documents
- Scaling software systems by an order of magnitude (pick your dimension)
- Supporting the web in a desktop productivity suite
- Adding support for Asian languages to a tool

One more difficulty in more depth

17

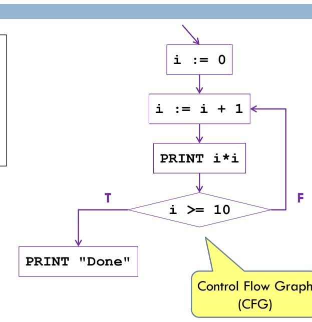
- Dijkstra's1968 "**goto** considered harmful" letter to the editor of CACM is a classic
  - Mark Twain: "A classic is something everyone wants to have read, but nobody wants to read."
- My version of his key argument is that
  - We write programs but we care about executions – getting the behaviors we want is indirect
  - But reasoning about arbitrary behaviors is very hard due to the limits of the human brain
  - By reducing the gap between the program and the behaviors, we can do better in terms of reasoning

Example

[Adapted from Wikipedia, Spaghetti Code]

18

```
10 i = 0
20 i = i + 1
30 PRINT i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Done"
70 END
```

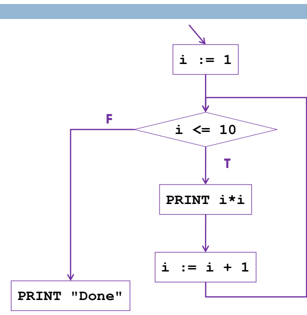


- Only one behavior in this example
- Not straightforward to reason about (at least in general) – must simulate the control flow

Example continued: use **while** loop

19

```
i := 1;
while i <= 10 do
    PRINT i*i;
    i := i + 1
end;
PRINT "DONE"
```



- Still only one behavior
- The loop is clearer – can more easily separate "doing the loop" and "exiting the loop"
- Will allow invariants and proofs – easier reasoning

Böhm and Jacopini ● CACM May 1966

[Wikipedia for additional history]

20

- They showed a construction that takes an arbitrary program and produces a program with equivalent behaviors that has a structured control flow graph that uses only
  - sequencing (**;**)
  - conditionals (**if-then**)
  - loops (**while-do**)
- Basic idea: encode the program counter in a program variable
- So, what's the problem?

Programming languages research

- *Very* roughly, (my view is that) most programming languages research focuses on ways to reason about sets of behaviors through programs
- One program with (most often) an unbounded numbers of behaviors
- Changes are to the program, with the intent of achieving desired changes in the behaviors

Proofs-of-correctness

- A strong connection between the static program and the dynamic behaviors also enables proofs-of-correctness to be done precisely and formally
- Dijkstra, Hoare, Wirth, et al. did this in the late 1960's and early 1970's as step-wise refinement
- Pseudo-code is repeatedly expanded until the translation into programming language code is obvious
  - Choose a module
  - Decompose into smaller modules
  - Repeat until all modules are easily understood
- Provide explicit specification of the program, annotate it with assertions, use programming language semantics to prove those assertions

Basics of proofs-of-correctness

- In a logic, write down the *specification*
  - the effect of the computation that the program is required to perform (the postcondition Q)
  - any constraints on the input environment to allow this computation (the precondition P)
- A Hoare triple is a predicate { P } `S` { Q} that is true whenever P holds and the execution of `S` guarantees that Q holds
- To prove { P } `S` { Q} requires
  - a precisely defined logical meaning for each construct in the programming language
  - insertion of intermediate assertions to allow proofs to "flow" through the program

{ P } S { Q }


```
{ P }
S1;
S2;
if (..)
  S3
else
  S4
fi;
S5
{ Q }
```

```
{ P }
S1;
{ A }
S2;
{ B }
if (..)
  S3
else
  S4
fi;
{ C }
S5
{ Q }
```

- Prove { P } **S1** { A }
- Prove { A } **S2** { B }
- Prove { B } **if…** { C }
- Prove { C } **S5** { Q }

Trivial examples

- { true }
  `y := x * x;`
  $\{y \geq 0\}$
- { x <> 0 }
  `y := x * x;`
  $\{ y > 0 \}$
- $\{ x > 0 \}$
  `x := x + 1;`
  $\{ x > 1 \}$
- $\{ x = k \}$
  ```
  if (x < 0)
    x := -x
  fi;
  ```
  { ? }
- { ? }
  `x := 3;`
  $\{ x = 8 \}$

The objective is the proof

[Example from Aldrich/Leino]

- Simply having true post-conditions is not sufficient
  - $\{ x = 5 \}$ `x := x * 2` { true }
  - $\{ x = 5 \}$ `x := x * 2` $\{ x > 0 \}$
  - $\{ x = 5 \}$ `x := x * 2` $\{ x = 10 \mid\mid x = 5 \}$
  - $\{ x = 5 \}$ `x := x * 2` $\{ x = 10 \}$
- It is generally important to look for the logically strongest post-condition – that is, one that represents the most restrictive assertion consistent with the specification or with intermediate assertions

Weakest preconditions

[Example from Aldrich/Leino]

Here are a number of valid Hoare Triples

- $\{x = 5 \;\&\&\; y = 10\}$ `z := x / y` $\{ z < 1 \}$
- $\{x < y \;\&\&\; y > 0\}$ `z := x / y` $\{ z < 1\}$
- $\{y \neq 0 \;\&\&\; x / y < 1\}$ `z := x / y` $\{ z < 1 \}$

- The last one is the most useful because it allows us to invoke the program in the most general condition – it is called the *weakest precondition*, `wp(S,Q)` of `S` with respect to `Q`
  - If {P} `S` {Q} and for all P' such that P' => P, then P is wp(S,Q)

Conditional execution


```
{P}          {true}
if C         if x >= y
  S1           max := x
else         else
  S2           max := y
fi           fi
{Q}          {(max >= x ∧ max >= y)}
```

A formal definition of the semantics of `if-then-else`

$\{P \wedge C\}$ `S1` $\{Q\} \wedge \{P \wedge \neg C\}$ `S2` $\{Q\}$

$(\max \geq x \wedge \max \geq y)$

29

- Is this a good post-condition?
- Does it do what we "want" or "expect"?

Formalism doesn't eliminate all confusion

30

- We likely want $(\max = x \vee \max = y) \wedge (\max \geq x \wedge \max \geq y)$
- In essence, every specification is satisfied by an infinite number of programs and vice versa
- Here's another potentially confusing post-condition
  - $\forall (i < j) \bullet A[i] \leq A[j]$
- Boehm's distinction in action: formalism is much more useful in showing you've built the system right than in showing you've built the right system

Assignment statements

31

- $\{Q(E)\}$ **x := E** $\{Q(x)\}$
- If we knew something to be true about **E** before the assignment, then we know it to be true about **x** after the assignment (assuming no side-effects)

$\{y > 0\}$
x := y
$\{x > 0\}$

$\{x > 0\}$
x := x + 1;
$\{x > 1\}$

$[Q(E) \equiv x+1 > 1 \equiv x > 0]$

$[Q(x) \equiv x > 1]$

Loops: $\{P\}$ **while B do S** $\{Q\}$

32

- We can try to unroll this into
  - $\{P \wedge \neg B\}$ S $\{Q\}$ $\vee$
    $\{P \wedge B\}$ S$\{Q \wedge \neg B\}$ $\vee$
    $\{P \wedge B\}$ S $\{Q \wedge B\}$ S $\{Q \wedge \neg B\}$ $\vee \ldots$
- But we don't know how far to unroll, since we don't know how many times the loop can execute
- The most common approach to this is to find a *loop invariant*, which is a predicate that
  - is true each time the loop head is reached (on entry and after each iteration)
  - and helps us prove the post-condition of the loop
- The loop invariant approximates the fixed point of the loop

Three steps: find I such that…

33

- $P \Rightarrow I$ –Invariant is correct on entry
- $\{B \wedge I\}$ S $\{I\}$ –Invariant is maintained

$\{\neg B \wedge I\} \Rightarrow Q$ –Q is true when loop terminates

$\{n > 0\}$
```
x := a[1];
i := 2;
while i <= n do
  if a[i] > x then x := a[i];
  i := i + 1;
end;
```
$\{x = \max(a[i],\ldots,a[n])\}$

OK, what's I?

503 11sp © UW CSE • D. Notkin

Termination

34

- Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper post-condition if it terminates – this is called *weak correctness*
- A Hoare triple for which termination has been proven is *strongly correct*
- Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets

503 11sp © UW CSE • D. Notkin

Coming up next week

35

- Proving properties of abstract data types
- Separate proofs of the specification (e.g., properties like $x = S.top(S.push(x))$ and of the concrete implementation of the methods (top, push, etc.)
- Define an *abstraction function* that gives a mapping from instances of the concrete representation to the abstract representation
- Define a *representation invariant* that holds across all legal instances of the concrete representation

503 11sp © UW CSE • D. Notkin

Open issues

36

- Automation – proof engines, proof assistants, etc.
- Programming language dimensions – side-effects, procedures/methods (and parameter passing), non-local control (e.g., exceptions), classes/objects etc., other language paradigms (e.g., functional), …
- Whence post-conditions?
- How much of a proof needs to be redone if the specification and/or the program changes slightly?
- …

503 11sp © UW CSE • D. Notkin

Programming languages

37

- Dijkstra's notion of structured control flow easing reasoning is completely rational
- At the same time, the "real" world of software has clearly decided that constructs that help them in their work seem to be more important than keeping a close connection between the static program and dynamic behaviors
  - Examples include exceptions, event-based programming, aspect-oriented programming, and more
- This leaves interesting research about how to improve reasoning in the face of constructs like these

So…

38

- …the original promise of program verification has not been achieved, at least to the degree many anticipated
- At the same time, as we'll see, it's clear that the underlying techniques have made a huge difference and have supported a shift from
  - trying to prove big theorems about little programs to
  - trying to prove little theorems about big programs
- Aside: type-checking is in the second category

Debunking a myth

39

- A culture – at least in the research world – developed in part due to this proof-centric view of the world
- Roughly, it is crucial to prove properties of programs over all possible executions – otherwise the other executions may have unexpected behaviors
  - That is, sampling of the behaviors ("testing") is inherently problematic

Sources of unsoundness:

Dwyer et al.

40

- Matt Dwyer's talk at ICSE 2007 put much of this issue in perspective: in my words, he argues that it's *all* sampling
- Dynamic techniques sample across executions (behaviors)
  - The "hope" is that some behaviors are characteristic of other behaviors
- Static techniques sample across properties (requirements)
  - The "hope" is that some requirements are good proxies for other requirements (e.g., type-safe and deadlock-free build confidence in correctness)
- What we need to know is the degree of unsoundness
  - That is, we need to know what we know, and what we don't know
- The following few slides are from Dwyer's talk

It's about coverage [from Dwyer]

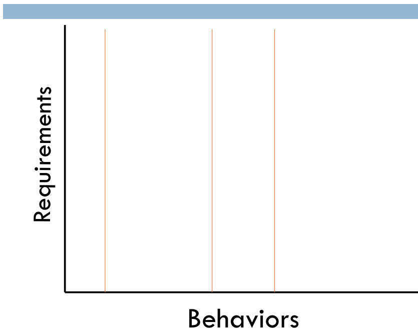
41

- Nobody believes that one technique will "do it all"
- A suite of techniques will be required
- If techniques involve sampling how do we know
  - They cover the breadth of software requirements
  - They cover the totality of program behavior

A unified theory of behavioral coverage is needed

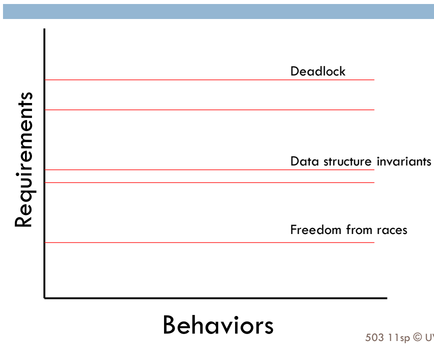
"Testing": sampling across behaviors

42



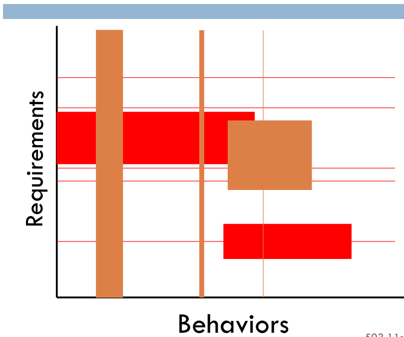
"Proving": sampling across properties

43



When can we broaden with confidence?

44



Static vs. dynamic techniques

45

- It really shouldn't be "versus"
- Each has strengths, each has weaknesses
- This is increasingly recognized in research – but not by everybody!

503 11sp © UW CSE • D. Notkin