Advanced Software Testing

Carmine Vassallo

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Forty seconds of Ariane-5 (1996)

Traditional Testing Techniques

- Black-box Testing
  - Test cases based on an analysis of the description of the product without reference to its internal workings
  - e.g., Equivalence partitioning, Boundary value analysis

- White-box Testing
  - Test cases based on an analysis of the internal structure of the component or system
  - e.g., Condition coverage, Path coverage

Let’s talk about advanced testing techniques

Fuzzing

Fuzz Testing (Fuzzing)

- Fuzz testing or Fuzzing is a Black Box software testing technique, which consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.

Fuzz Testing (Fuzzing)

- Let’s consider an integer in a program, which stores the result of a user's choice between 3 questions.
- When the user picks one, the choice will be 0, 1 or 2.
- But what if we transmit 3, or 255 ?
  - the program may crash and lead to "classical" security issues: (un)exploitable buffer overflows, DoS, ...

What can be fuzzed

- Parser of any kind (xml, json, pdf, truetype, …)
- Media codecs (audio, video, raster & vector images, etc)
- Network protocols (HTTP, RPC, SMTP, MIME...)
- Crypto (boringssl, openssl)
- Compression (zip, gzip, bzip2, brotli, ...)
- Formatted output (sprintf, template engines)
- Compilers and interpreters (Javascript, PHP, Perl, Python, Go, Clang, ...)
- Regular expression matchers (PCRE, RE2, libc's regcomp)
- Text/UTF processing (icu)
- Databases (SQLite)
- Browsers, text editors/processors (Chrome, vim, OpenOffice)
- OS Kernels (Linux), drivers, supervisors and VMs

Fuzzer

- A fuzzer is a program which injects automatically semi-random data into a program/stack and detect bugs
- The data-generation part is made of generators, and vulnerability identification relies on debugging tools

Type of Fuzzers

- Grammar-based generation: generate random inputs according to grammar rules
  - Peach, packetdrill, csmith, gosmith, syzkaller
- Blind mutation: requires a corpus of representative inputs, apply random mutations to them
  - ZZUF, Radamsa
- Grammar reverse-engineering: learn grammar from existing inputs using algorithmic approach of machine learning
  - Sequitur algorithm, go-fuzz
- Symbolic execution + SAT solver: synthesize inputs with maximum coverage using black magic
  - KLEE
- Coverage-guided fuzzers: genetic algorithm that strives to maximize code coverage
  - libFuzzer, AFL, honggfuzz, syzkalle
- Hybrid

How Fuzz Testing works

```
int main() {

    int a[2] = {1, 0};

    int b=a[2];

}
```

- *off-by-one error*
- The variable b will end up containing an arbitrary value.
- Recent versions of the compilers llvm and gcc got a powerful tool to spot such memory access bugs: *Address Sanitizer (ASan)*

How Fuzz Testing works

- gcc -fsanitize=address -ggdb -o test test.c

```
==7402==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff2971ab88 at pc 0x400904 bp 0x7fff2971ab40 sp 0x7fff2971ab30

READ of size 4 at 0x7fff2971ab88 thread T0

    #0 0x400903 in main /tmp/test.c:3

    #1 0x7fd7e2601f9f in __libc_start_main (/lib64/libc.so.6+0x1ff9f)

    #2 0x400778 (/tmp/a.out+0x400778)


Address 0x7fff2971ab88 is located in stack of thread T0 at offset 40 in frame

    #0 0x400855 in main /tmp/test.c:1


  This frame has 1 object(s):

    [32, 40) 'a' <== Memory access at offset 40 overflows this variable

HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext

      (longjmp and C++ exceptions *are* supported)

SUMMARY: AddressSanitizer: stack-buffer-overflow /tmp/test.c:3 main
```

Sanitizers

- AddressSanitizer (detects addressability issues)
- LeakSanitizer (detects memory leaks)
- ThreadSanitizer (detects data races and deadlocks) for C++ and Go
- MemorySanitizer (detects use of uninitialized memory)

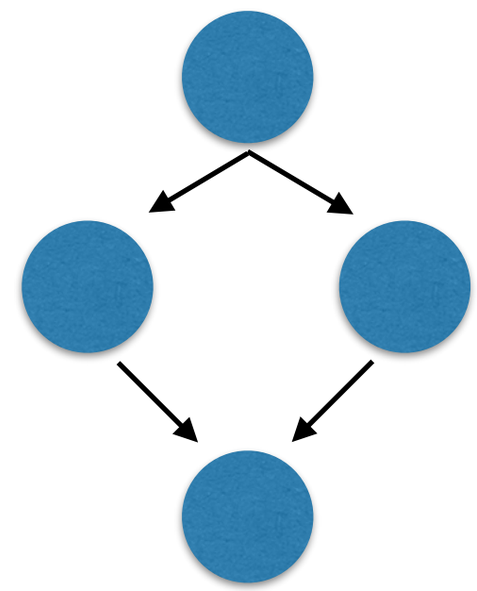
Data Flow Analysis

Data Flow Analysis (DFA)

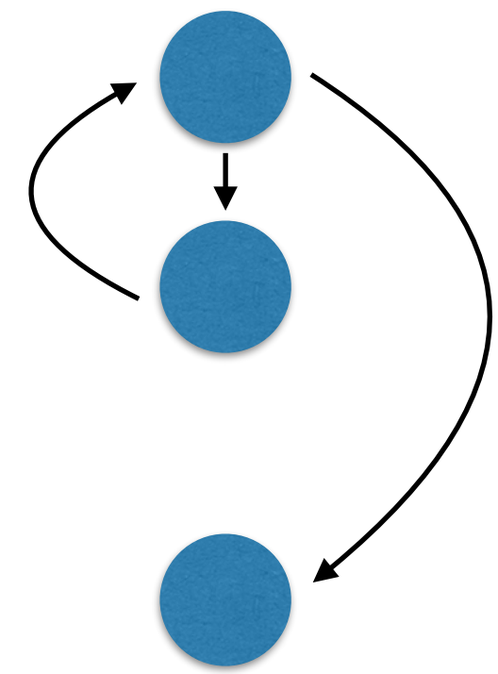
- Data-flow analysis is a white-box testing technique aiming at detecting anomalies during the evolution of variables in the source code
- These anomalies could be symptoms of bugs
- A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate

Control Flow Graph (CFG)

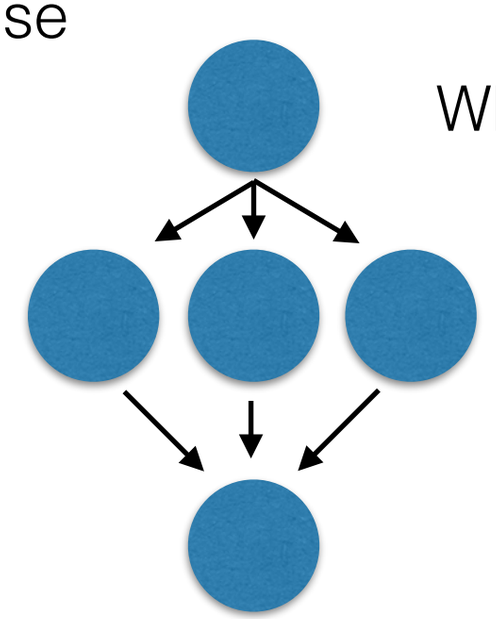
- Directed graph
- Nodes are blocks of sequential statements
- Edges are transfers of control
- Edges may be labeled with predicate representing the condition of control transfer



If-then-else



While loop



Switch

Type of variable occurrences

- Definition occurrence (**d**): value is bound to variable
- Use occurrence (**u**): value of variable is referred
- Cancellation occurrence (**c**): value of variable is meaningless

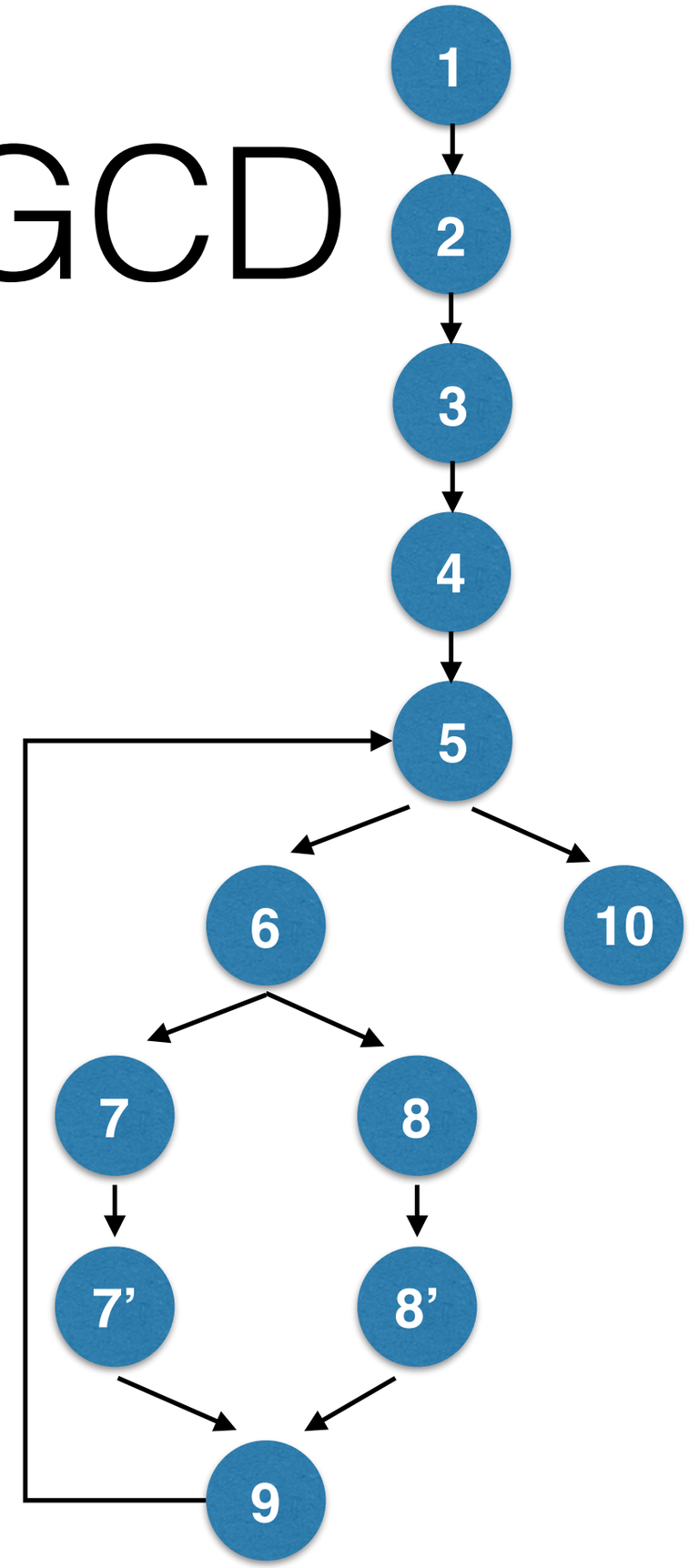
How to detect anomalies?

- Basic rules
  - *RULE 1*: ***u*** has to be preceded by ***d*** without intermediate ***a***
  - *RULE 2*: ***d*** has to be followed by ***u*** before a new ***d***

DFA Example: GCD

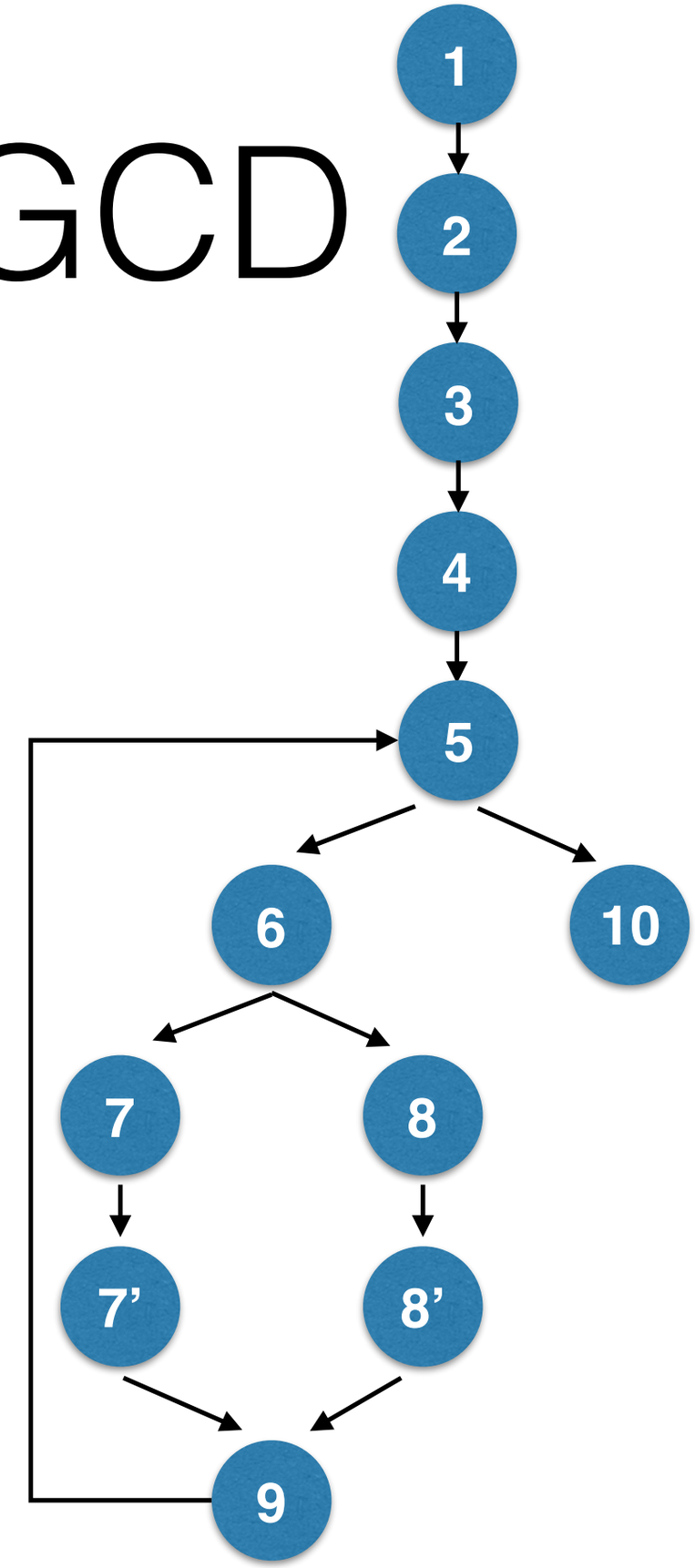
- Greatest common divisor (GCD)

```
1.  int x,y,a,b;
2.  read(x,y);
3.  a = x;
4.  a = y;
5.  while (a != b) {
6.          if (a > b) {
7.              a = a - b;}
8.          else {b = b - a;}
9.  }
10.System.out.println("GCD is: " + a);
```



DFA Example: GCD

Node	d	u	c
1			x,y,a,b
2	x,y		
3	a	x	
4	a	y	
5		a,b	
6		a,b	
7		a,b	
7’	a		
8		a,b	
8’	b		
10		a	



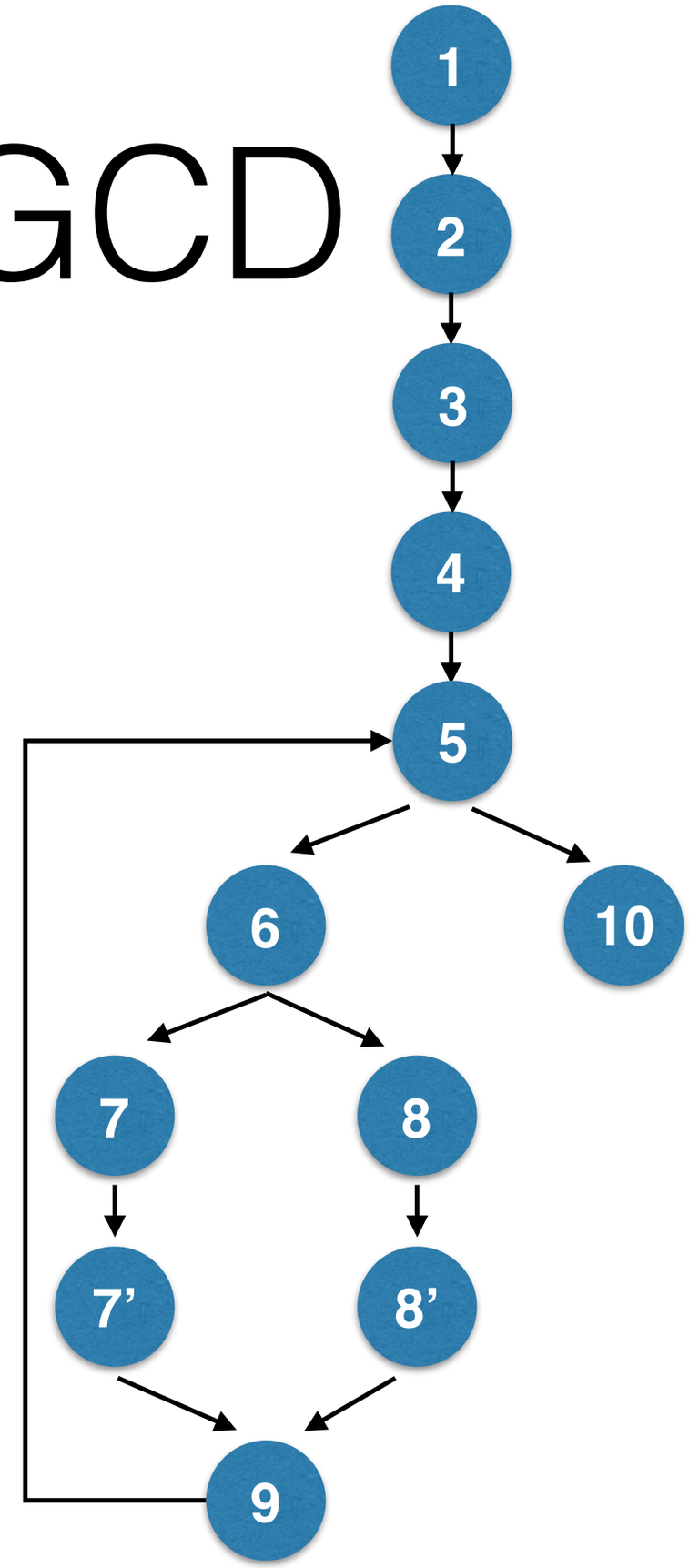
Evolution of “a”: c-**dd**uuudu-u

Evolution of “b”: **c**---**u**uu-u-d

DFA Example: GCD

- Greatest common divisor (GCD)

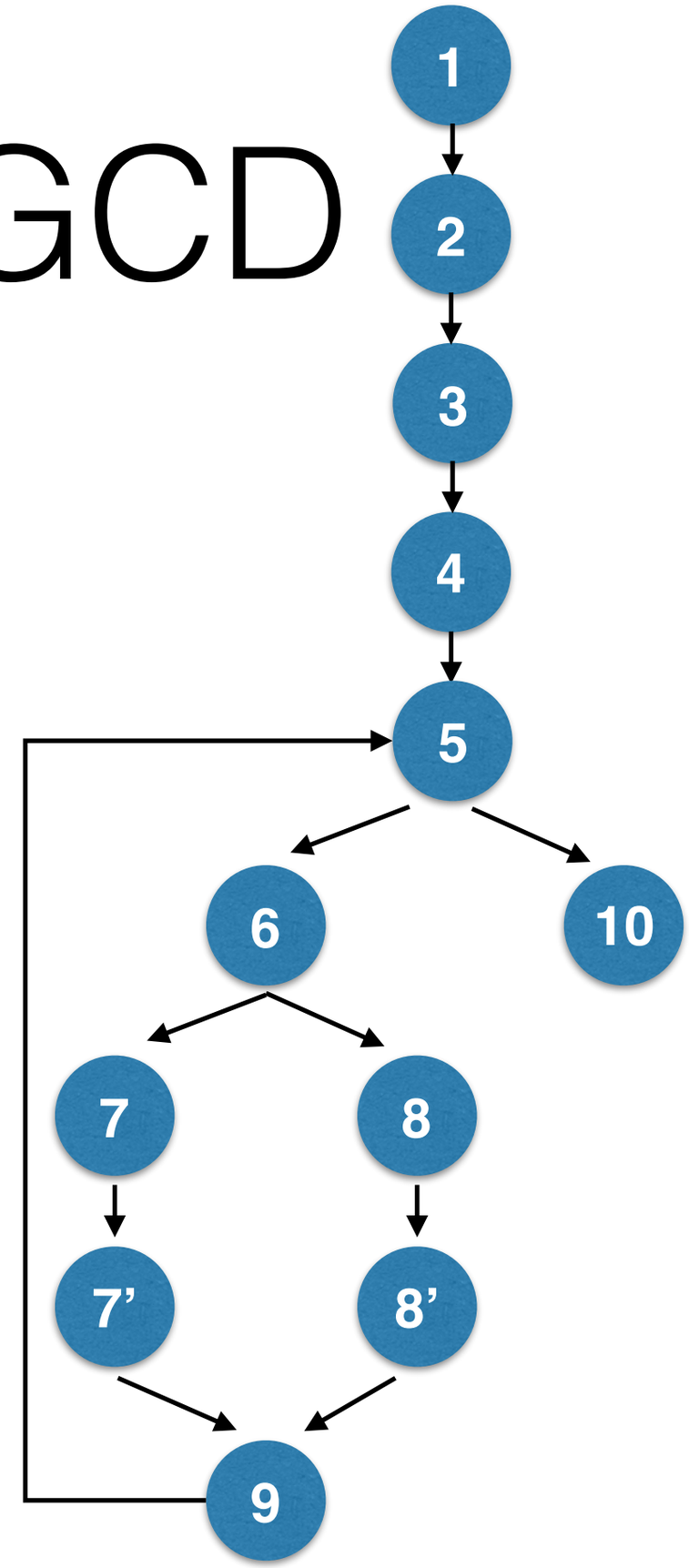
```
1.  int x,y,a,b;
2.  read(x,y);
3.  a = x;
4.  a = y;
5.  while (a != b) {
6.          if (a > b) {
7.              a = a - b;}
8.          else {b = b - a;}
9.  }
10. System.out.println("GCD is: " + a);
```



DFA Example: GCD

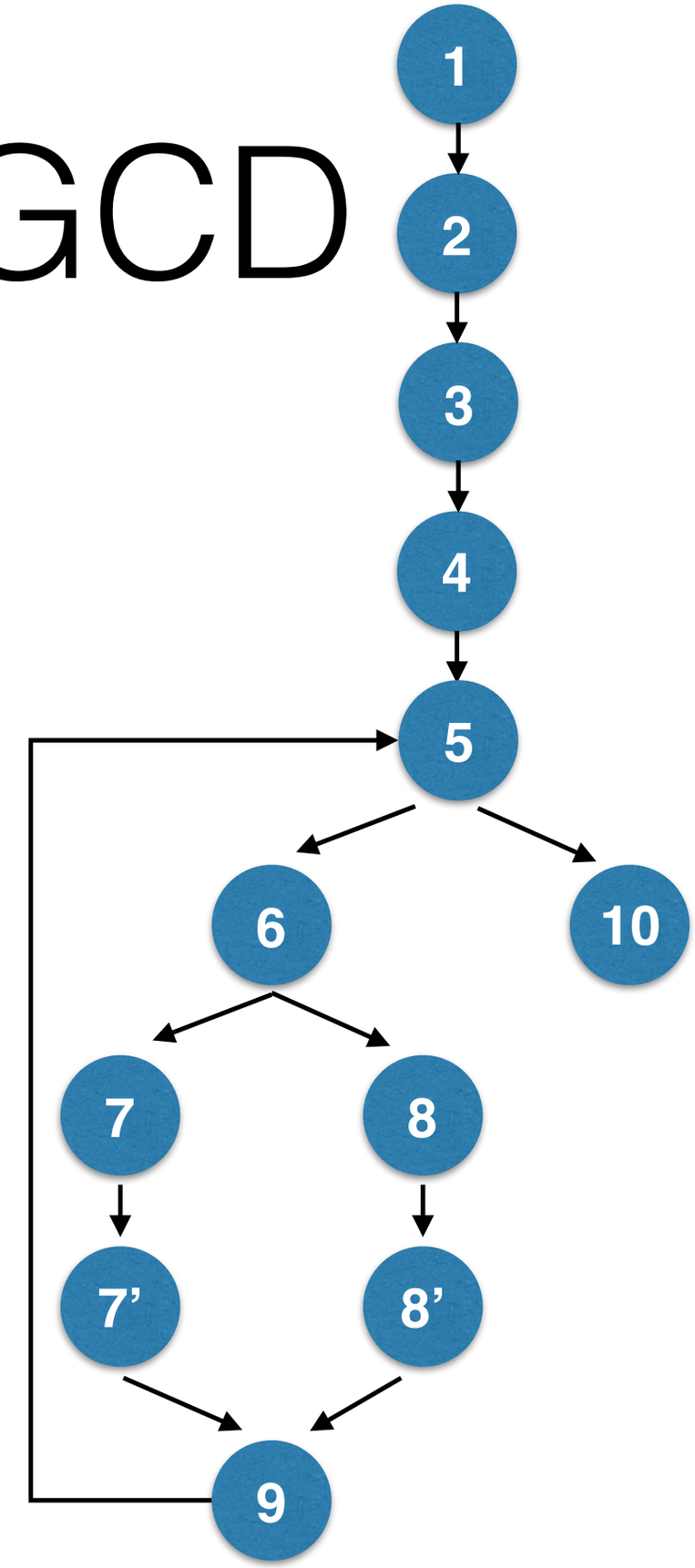
- Greatest common divisor (GCD)

```
1.  int x,y,a,b;
2.  read(x,y);
3.  a = x;
4.  b = y;
5.  while (a != b) {
6.          if (a > b) {
7.              a = a - b;}
8.          else {b = b - a;}
9.  }
10. System.out.println("GCD is: " + a);
```



DFA Example: GCD

Node	d	u	c
1			x,y,a,b
2	x,y		
3	a	x	
4	b	y	
5		a,b	
6		a,b	
7		a,b	
7’	a		
8		a,b	
8’	b		
10		a	



Evolution of “a”: c-**du**uuudu-u

Evolution of “b”: **c**--**du**uu-u-d

What about *testing* the test cases?

Mutation Testing

- It aims at assessing the effectiveness of the prepared test cases, rather than actually looking for defects
- Basic Idea:
  - Take a program and test data generated for that program (using another test technique)
  - Create a number of similar programs (**mutants**), each differing from the original in one small way, i.e., each possessing a fault
  - The original test data are run through the mutants
  - If test data detect differences in mutants, then the mutants are said to be **dead**, and the **test set is adequate**

Mutation Testing

- A mutant remains **alive** either because it is equivalent to the original program (equivalent mutant - functionally identical although syntactically different) or the **test set is inadequate** to kill the mutant

- For automated generation of mutants, **mutation operators** are used

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice, but the number will increase with languages

Mutation Operators

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement

- Specific to object-oriented programming languages:
  - Replacing a type with a compatible type (inheritance)
  - Changing the access modifier of an attribute, a method
  - Changing the instance creation expression (inheritance)
  - Changing the order of parameters in the definition of a method
  - Changing the order of parameters in a call
  - Removing an overloading method
  - Reducing the number of parameters
  - Removing an overriding method
  - Removing a hiding field
  - Adding a hiding field

Mutation Coverage

- Complete coverage equate to killing all non-equivalent mutants
- The amount of coverage is called “mutation score”
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax of the software
- Numbers of mutants tend to be large, even for small programs
- Tool: MuJava (http://cs.gmu.edu/~offutt/mujava)

Mutation Testing Example

The program prompts the user for a positive integer in the range 1 to 20 and then for a string of that length.The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string. The user has the option to search for more characters.

Code Chunk and Test Suite

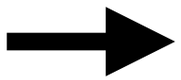
```
...
found := FALSE;
index := 0;
while(not(found)) and
(index < length) do begin

if word[index] = character
then found := TRUE

else

index := index + 1

end

...
```

Test Case	Length	Word	Character	Output	Expected Output
TC1	25			ERROR	ERROR
TC2	1	X	X	0	0
TC3	1	X	a	NOT FOUND	NOT FOUND

All test cases pass!

Mutant 1

```
...
found := FALSE;
index := 0;
while(not(found)) and (index < length) do
begin

if word[index] = character then found :=
TRUE

else

index := index + 1

end

...
```



```
...
found := TRUE;
index := 0;
while(not(found)) and (index < length) do
begin

if word[index] = character then found :=
TRUE

else

index := index + 1

end

...
```

Mutant 1: Testing

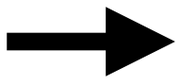
```
...
found := TRUE;
index := 0;
while(not(found)) and
(index < length) do begin

if word[index] = character
then found := TRUE

else

index := index + 1

end

...
```

Test Case	Length	Word	Character	Output	Expected Output
TC1	25			ERROR	ERROR
TC2	1	X	X	0	0
TC3	1	X	a	**0**	**NOT FOUND**

TC3 kills the Mutant !

Mutant 2

```
...
found := FALSE;
index := 0;
while(not(found)) and (index < length) do
begin

if word[index] = character then found :=
TRUE

else

index := index + 1

end

...
```



```
...
found := FALSE;
length := 1;
while(not(found)) and (index < length) do
begin

if word[index] = character then found :=
TRUE

else

index := index + 1

end

...
```

Mutant 2: Testing

```
...
found := TRUE;
length := 1;
while(not(found)) and
(index < length) do begin

if word[index] = character
then found := TRUE

else

index := index + 1

end

...
```

Test Case	Length	Word	Character	Output	Expected Output
TC1	25			ERROR	ERROR
TC2	1	X	X	0	0
TC3	1	X	a	-1	-1

The Mutant is still alive,
no tests detect it !
Let's modify our test suite!

Mutant 2: Testing

```
...
found := TRUE;
length := 1;
while(not(found)) and
(index < length) do begin

if word[index] = character
then found := TRUE

else

index := index + 1

end

...
```

Test Case	Length	Word	Character	Output	Expected Output
TC1	25			ERROR	ERROR
TC2	5	Xenon	X	0	0
TC3	5	Xenon	a	-1	-1
TC4	5	Xenon	n	**-1**	**2**

TC4 kills the Mutant!

Mutation Testing

- It measures the quality of test cases
- It provides the tester with a clear target (mutants to kill)
- Mutation testing shows certain kinds of faults (specified by the fault model) are unlikely
- It does force the programmer to scrutinize the code and think of the test data that will expose certain kinds of faults
- Computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Probably most useful at unit testing level
- Some systems have been designed to help but still time consuming

Faults are sly!

The client was a major political party about to use the platform for communication purposes on elections day.

What could possibly go wrong, right?

When launched, the system simply didn't work, and it being a Sunday meant that the team wasn't readily available.

What caused the problem?

The fact that it was Sunday. And that the tests had only ran from Monday to Friday. As it turns out, the platform had a bug that rendered it inoperable during Sundays (something to do with 0-indexed arrays).

That’s where a human comes in, analysing code with a critical mindset and thinking of possibilities that may have been neglected when the tests were written.

In retrospect, Code Review could have indeed saved the day, as analysing the code with a thorough attitude could have caught this particular flaw.

That’s where a human comes in, analysing code with a critical mindset and thinking of possibilities that may have been neglected when the tests were written.

In retrospect, Code Review could have indeed saved the day, as analysing the code with a thorough attitude could have caught this particular flaw.

Code Review

- A manual inspection of source code by developers other than the author
- In the past, code review was very formal
  - line-by-line group reviews, done in extended meetings
- Nowadays many organizations (e.g., Microsoft, Google, Facebook) are adopting an informal and tool-based code review integrated in the Continuous Integration process

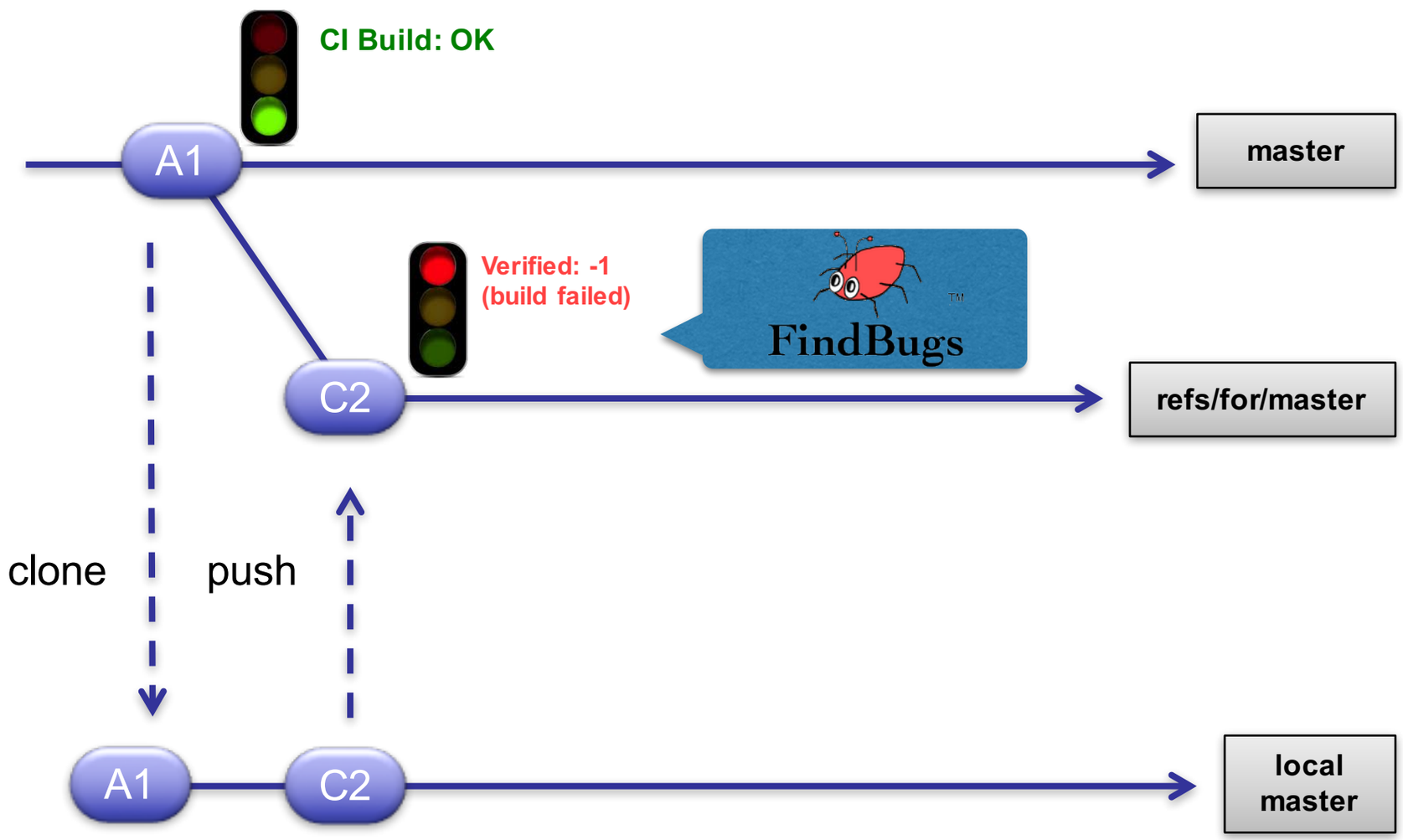
Modern Code Review

- Tool-based

Code Review with Gerrit

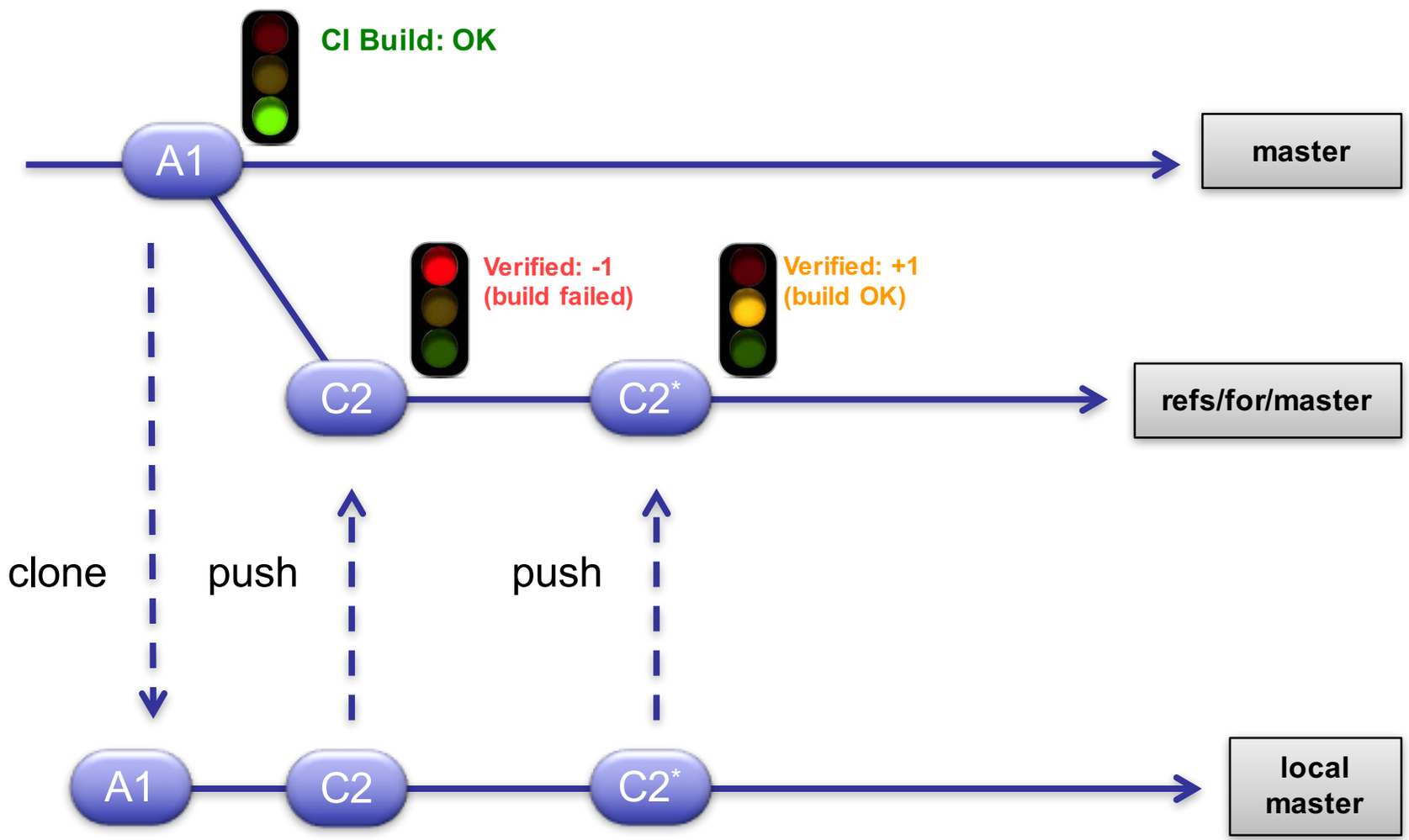
Code Review with Gerrit

Code Review with Gerrit



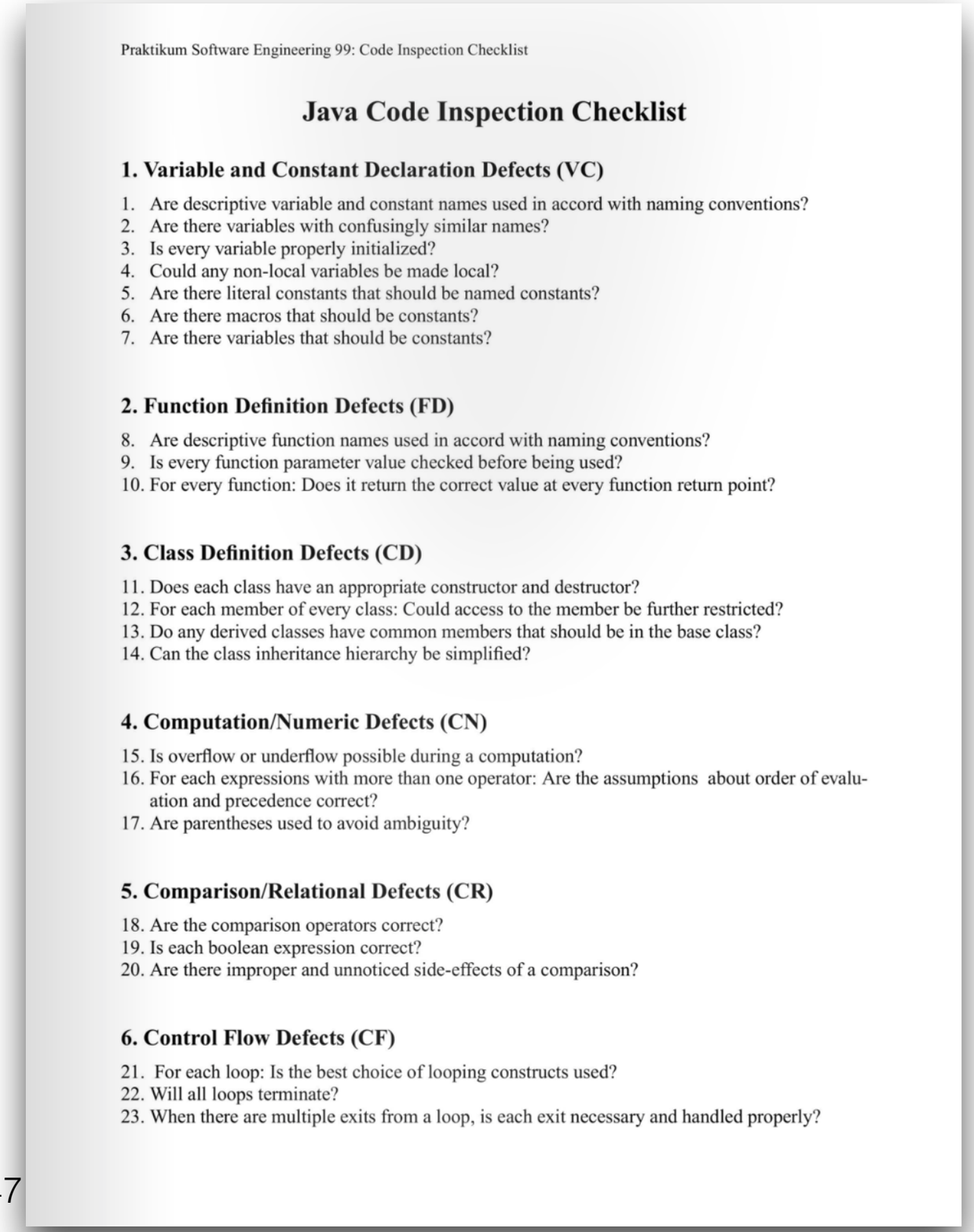
Build could fail because of detection of bug patterns (through static analysis tools, e.g., FindBugs)

Code Review with Gerrit



Code Review with Gerrit

- Once the project is built correctly, the proposed change is inspected by reviewers
- While searching for defects reviewers is supported by a checklist (other than <u>his experience</u>)



Praktikum Software Engineering 99: Code Inspection Checklist

Java Code Inspection Checklist

1. Variable and Constant Declaration Defects (VC)

1. Are descriptive variable and constant names used in accord with naming conventions?
2. Are there variables with confusingly similar names?
3. Is every variable properly initialized?
4. Could any non-local variables be made local?
5. Are there literal constants that should be named constants?
6. Are there macros that should be constants?
7. Are there variables that should be constants?

2. Function Definition Defects (FD)

8. Are descriptive function names used in accord with naming conventions?
9. Is every function parameter value checked before being used?
10. For every function: Does it return the correct value at every function return point?

3. Class Definition Defects (CD)

11. Does each class have an appropriate constructor and destructor?
12. For each member of every class: Could access to the member be further restricted?
13. Do any derived classes have common members that should be in the base class?
14. Can the class inheritance hierarchy be simplified?

4. Computation/Numeric Defects (CN)

15. Is overflow or underflow possible during a computation?
16. For each expressions with more than one operator: Are the assumptions about order of evaluation and precedence correct?
17. Are parentheses used to avoid ambiguity?

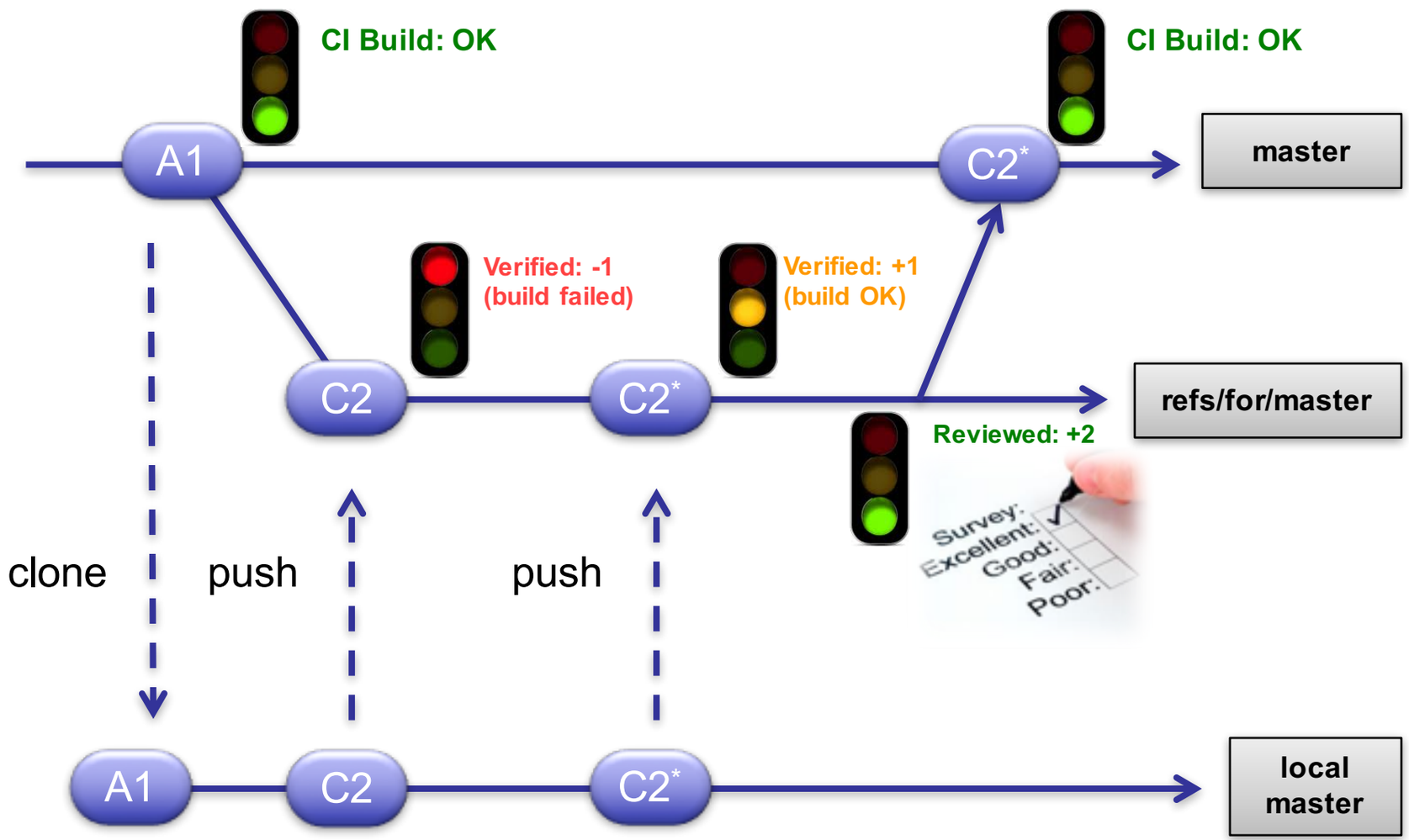
5. Comparison/Relational Defects (CR)

18. Are the comparison operators correct?
19. Is each boolean expression correct?
20. Are there improper and unnoticed side-effects of a comparison?

6. Control Flow Defects (CF)

21. For each loop: Is the best choice of looping constructs used?
22. Will all loops terminate?
23. When there are multiple exits from a loop, is each exit necessary and handled properly?

Code Review with Gerrit



Gerrit: Eclipse CDT

eclipse

Change-Id:	I0fcbad27155d875b420ce99cd9e9ba202448cc59
Owner	Nathan Ridge
Project	cdt/org.eclipse.cdt
Branch	master
Topic	
Uploaded	May 13, 2014 8:07 AM
Updated	May 14, 2014 7:03 PM
Status	**Merged**

Commit Message — Permalink

```
Bug 432701 - Move code that determines the value of an initializer to SemanticUtil

Change-Id: I0fcbad27155d875b420ce99cd9e9ba202448cc59
Signed-off-by: Nathan Ridge <zeratul976@hotmail.com>
Reviewed-on: https://git.eclipse.org/r/26418
Reviewed-by: Sergey Prigogin <eclipse.sprigogin@gmail.com>
Tested-by: Sergey Prigogin <eclipse.sprigogin@gmail.com>
```

Reviewer	Code-Review	Verified
Nathan Ridge		
Hudson CI		
Sergey Prigogin	✓	✓

Review Scores

▶ **Included in**
▶ **Dependencies**

Reference Version: Base

▶ **Patch Set 1** 5d1344ff41e40638e5bfda52139e0b6c0b3c69f9
▼ **Patch Set 2** 6a11b957b02fae5c58a4fe5cff15cc949939d13e

Patch Sets

Files Modified (or Commits)

Author	Nathan Ridge <zeratul976@hotmail.com> May 14, 2014 9:15 AM
Committer	Nathan Ridge <zeratul976@hotmail.com> May 14, 2014 9:36 AM
Parent(s)	08f5b7e52a2218704be93dce79eddb6f779eb33e Bug 415486 - Make the indexer prioritize the files that are currently open ...
Download	checkout | pull | cherry-pick | patch | Anonymous Git | Anonymous HTTP |

```
git fetch git://git.eclipse.org/gitroot/cdt/org.eclipse.cdt refs/changes/18/26418/2 && git checkout FETCH_HEAD
```

	File Path	Comments	Size	Diff	
▸	Commit Message			Side-by-Side	Unified
M	core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/dom/parser/cpp/CPPParameter.java		+5, -3	Side-by-Side	Unified
M	core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/dom/parser/cpp/CPPVariable.java		+1, -31	Side-by-Side	Unified
M	core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/dom/parser/cpp/semantics/SemanticUtil.java		+41, -1	Side-by-Side	Unified
M	qt/org.eclipse.cdt.qt.ui/src/org/eclipse/cdt/internal/qt/ui/assist/QPropertyAttributeProposal.java		+1, -1	Side-by-Side	Unified
			+48, -36	All Side-by-Side	All Unified

▶ **Patch Set 3** 0f45a9b6dfb3523db209abe229bb6f13f702a6e6
▶ **Patch Set 4** 6cbdabfbec623f01291a994f1bd76bbb9fc42248

Reviewers/Verifiers Comments

Comments — Expand Recent | Expand All | Collapse All

Nathan Ridge	Uploaded patch set 1.	May 13 8:07 AM
Hudson CI	Patch Set 1: Build Started ...	May 13 8:07 AM
Nathan Ridge	Patch Set 1: (2 comments) Please see the comments for a couple of things ...	May 13 8:13 AM

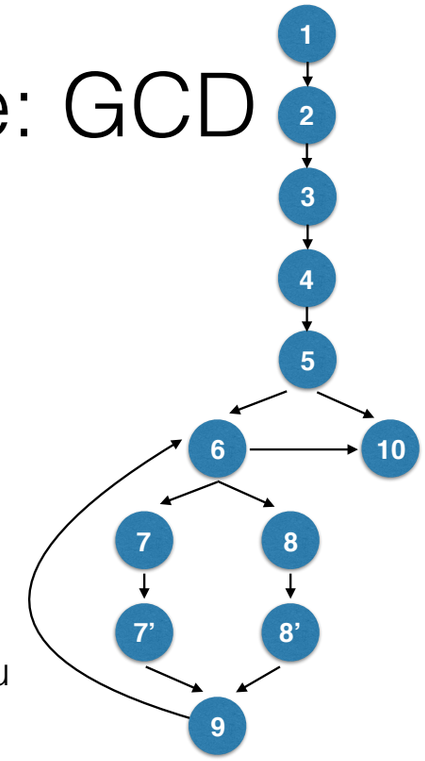
How Fuzz Testing works

```
int main() {

    int a[2] = {1, 0};

    int b=a[2];

}
```

- *off-by-one error*
- The variable b will end up containing an arbitrary value.
- Recent versions of the compilers llvm and gcc got a powerful tool to spot such memory access bugs: *Address Sanitizer (ASan)*

DFA Example: GCD

Node	d	u	c
1			x,y,a,b
2	x,y		
3	a	x	
4	a	y	
5		a,b	
6		a,b	
7		a,b	
7’	a		
8		a,b	
8’	b		
10		a	



Evolution of “a”: c-**dd**uuudu-u
Evolution of “b”: **c**---**u**uu-u-d

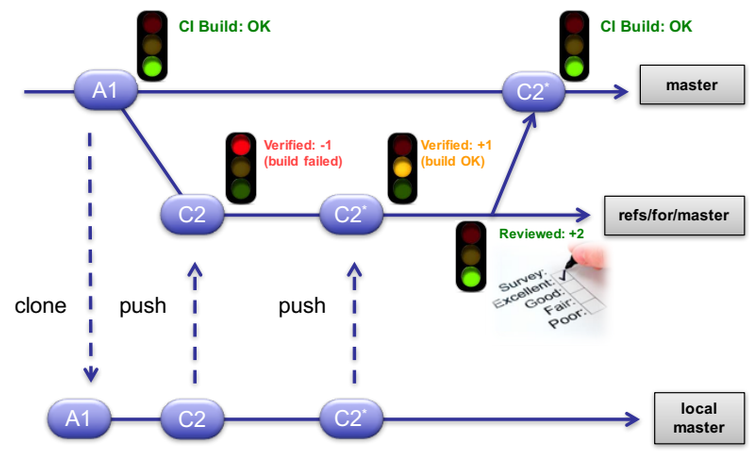
Mutant 1: Testing

```
...
found := TRUE;
index := 1;
while(not(found)) and
(index <= length) do
begin

if word[index] = character
then found := TRUE

else

index := index + 1

end

...
```

Test Case	Length	Word	Character	Output	Expected Output
TC1	25			ERROR	ERROR
TC2	1	X	X	0	0
TC3	1	X	a	**0**	**NOT FOUND**

TC3 kills the Mutant !

Code Review with Gerrit



References

- Paul Ammann and Jeff Offutt, Introduction to Software Testing (2nd Edition), Cambridge University Press
- Anne Mette Jonassen Hass, Guide to Advanced Software Testing, Artech House
- https://blog.codacy.com/code-review-vs-testing-804f52fd6553
- https://www.slideshare.net/sebastianopanichella/saner2015-v2
- https://fuzzing-project.org/tutorial2.html
- https://www.owasp.org/index.php/Fuzzing#Fuzzers_from_OWASP
- http://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html?page=3
- https://www.yumpu.com/en/document/view/30815982/java-code-inspection-checklist-undergraduate
- https://www.slideshare.net/DmitryVyukov/fuzzing-the-new-unit-testing?qid=c1481faa-c849-47e6-9103-e6fb8f02c5d5&v=&b=&from_search=1
- https://www.slideshare.net/lucamilanesio/gerrit-code-review